AAString-class  AAString objects

Description

An AAString object allows efficient storage and manipulation of a long amino acid sequence.

Details

The AAString class is a direct XString subtype (with no additional slot). Therefore all functions and methods described in the XString man page also work with an AAString object (inheritance). Unlike the BString container that allows storage of any single string (based on a single-byte character set) the AAString container can only store a string based on the Amino Acid alphabet (see below).

The Amino Acid alphabet

This alphabet contains all letters from the Single-Letter Amino Acid Code (see ?AMINO_ACID_CODE) + the stop ("*") the gap ("-"), and the hard masking ("+"), letters. It is stored in the AA_ALPHABET constant (character vector). The alphabet method also returns AA_ALPHABET when applied to an AAString object and is provided for convenience only.

Constructor-like functions and generics

In the code snippet below, x can be a single string (character vector of length 1) or a BString object.

AAString(x, start=1, nchar=NA, check=TRUE): Tries to convert x into an AAS-tring object by reading nchar letters starting at position start in x.

Accessor methods

In the code snippet below, x is an AAString object.

alphabet(x): If x is an AAString object, then return the Amino Acid alphabet (see above). See the corresponding man pages when x is a BString, DNAString or RNAString object.

Author(s)

H. Pages
See Also

AMINO_ACID_CODE, letter, XString-class, alphabetFrequency

Examples

```r
AA_ALPHABET
a <- AAString("MARKSEMSIR*")
length(a)
alphabet(a)
```

**AMINO_ACID_CODE**  
*The Single-Letter Amino Acid Code*

Description

Named character vector mapping single-letter amino acid representations to 3-letter amino acid representations.

See Also

AAString, GENETIC_CODE

Examples

```r
## See all the 3-letter codes
AMINO_ACID_CODE

## Convert an AAString object to a vector of 3-letter amino acid codes
aa <- AAString("LANDEECQW")
AMINO_ACID_CODE[strsplit(as.character(aa), NULL)[[1]]]
```

**AlignedXStringSet-class**  
*AlignedXStringSet and QualityAlignedXStringSet objects*

Description

The `AlignedXStringSet` and `QualityAlignedXStringSet` classes are containers for storing an aligned `XStringSet`.

Details

Before we define the notion of alignment, we introduce the notion of "filled-with-gaps subsequence". A "filled-with-gaps subsequence" of a string `string1` is obtained by inserting 0 or any number of gaps in a subsequence of `s1`. For example `L-A--ND` and `A--N-D` are "filled-with-gaps subsequences" of `LAND`. An alignment between two strings `string1` and `string2` results in two strings (align1 and align2) that have the same length and are "filled-with-gaps subsequences" of `string1` and `string2`.

For example, this is an alignment between `LAND` and `LEAVES`:
An alignment can be seen as a compact representation of one set of basic operations that transforms string1 into align1. There are 3 different kinds of basic operations: "insertions" (gaps in align1), "deletions" (gaps in align2), "replacements". The above alignment represents the following basic operations:

- insert E at pos 2
- insert V at pos 4
- insert E at pos 5
- replace by S at pos 6 (N is replaced by S)
- delete at pos 7 (D is deleted)

Note that "insert X at pos i" means that all letters at a position >= i are moved 1 place to the right before X is actually inserted.

There are many possible alignments between two given strings string1 and string2 and a common problem is to find the one (or those ones) with the highest score, i.e. with the lower total cost in terms of basic operations.

### Accesor methods

In the code snippets below, x is a `AlignedXStringSet` object.

- `unaligned(x)`: The original string.
- `aligned(x)`: The "filled-with-gaps subsequence" representing the aligned substring.
- `start(x)`: The start of the aligned substring.
- `end(x)`: The end of the aligned substring.
- `width(x)`: The width of the aligned substring, ignoring gaps.
- `indel(x)`: The positions, in the form of an `IRanges` object, of the insertions or deletions (depending on what the `AlignedXStringSet` object represents).
- `nindel(x)`: A two-column matrix containing the length and sum of the widths for each of the elements returned by `indel`.
- `length(x)`: The length of the `aligned(x)`.
- `nchar(x)`: The `nchar` of the `aligned(x)`.
- `alphabet(x)`: Equivalent to `alphabet(unaligned(x))`.
- `as.character(x)`: Converts `aligned(x)` to a character vector.
- `toString(x)`: Equivalent to `toString(as.character(x))`.

### Subsetting methods

- `x[i]`: Returns a new `AlignedXStringSet` object made of the selected elements.
- `rep(x, times)`: Returns a new `AlignedXStringSet` object made of the repeated elements.

### Author(s)

P. Aboyoun and H. Pages
DNAString-class

See Also

`pairwiseAlignment`, `PairwiseAlignedFixedSubject-class`, `XStringSet-class`

Examples

```r
pattern <- AString("LAND")
subject <- AString("LEAVES")
nw1 <- pairwiseAlignment(pattern, subject, substitutionMatrix = "BLOSUM50", gapOpening = -3, gapExtension = -1)
alignedPattern <- pattern(nw1)
unaligned(alignedPattern)
aligned(alignedPattern)
as.character(alignedPattern)
nchar(alignedPattern)
```

BOC_SubjectString-class

*BOC_SubjectString* and *BOC2_SubjectString* objects

Description

The BOC_SubjectString and BOC2_SubjectString classes are experimental and might not work properly.

Please DO NOT TRY TO USE them for now. Thanks for your comprehension!

Author(s)

H. Pages

DNAString-class

DNAString objects

Description

A DNAString object allows efficient storage and manipulation of a long DNA sequence.

Details

The DNAString class is a direct `XString` subtype (with no additional slot). Therefore all functions and methods described in the `XString` man page also work with a DNAString object (inheritance). Unlike the `BString` container that allows storage of any single string (based on a single-byte character set) the DNAString container can only store a string based on the DNA alphabet (see below). In addition, the letters stored in a DNAString object are encoded in a way that optimizes fast search algorithms.

The DNA alphabet

This alphabet contains all letters from the IUPAC Extended Genetic Alphabet (see `?IUPAC_CODE_MAP`) + the gap ("-" ) and the hard masking ("+") letters. It is stored in the `DNA_ALPHABET` constant (character vector). The `alphabet` method also returns `DNA_ALPHABET` when applied to a DNAString object and is provided for convenience only.
Constructor-like functions and generics

In the code snippet below, \( x \) can be a single string (character vector of length 1), a \texttt{BString} object or an \texttt{RNAString} object.

\[
\text{DNAString}(x, \text{start}=1, \text{nchar}=\text{NA}, \text{check}=\text{TRUE})\text{: Tries to convert } x \text{ into a DNAString object by reading } nchar \text{ letters starting at position start in } x.
\]

Accessor methods

In the code snippet below, \( x \) is a DNAString object.

\[
\text{alphabet}(x): \text{ If } x \text{ is a DNAString object, then return the DNA alphabet (see above). See the corresponding man pages when } x \text{ is a BString, RNAString or AAString object.}
\]

Author(s)

H. Pages

See Also

\texttt{IUPAC_CODE\_MAP}, \texttt{letter}, \texttt{XString-class}, \texttt{RNAString-class}, \texttt{reverseComplement}, \texttt{alphabetFrequency}

Examples

\[
\begin{align*}
\text{DNA\_BASES} \\
\text{DNA\_ALPHABET} \\
\text{d} & \leftarrow \text{DNAString}("\text{TTGAAAA\textendashCTC\textendashN}\text{")} \\
\text{length}(d) \\
\text{alphabet}(d) & \# \text{ DNA\_ALPHABET}
\end{align*}
\]

---

<table>
<thead>
<tr>
<th>GENETIC_CODE</th>
<th>The Standard Genetic Code</th>
</tr>
</thead>
</table>

Description

Two predefined objects (\texttt{GENETIC\_CODE} and \texttt{RNA\_GENETIC\_CODE}) that represent The Standard Genetic Code.

Usage

\texttt{GENETIC\_CODE} \newline
\texttt{RNA\_GENETIC\_CODE}

Details

Formally, a genetic code is a mapping between tri-nucleotide sequences called codons, and amino acids.

The Standard Genetic Code (aka The Canonical Genetic Code, or simply The Genetic Code) is the particular mapping that encodes the vast majority of genes in nature.

\texttt{GENETIC\_CODE} and \texttt{RNA\_GENETIC\_CODE} are predefined named character vectors that represent this mapping.
**Value**

`GENETIC_CODE` and `RNA_GENETIC_CODE` are both named character vectors of length 64 (the number of all possible tri-nucleotide sequences) where each element is a single letter representing either an amino acid or the stop codon "*" (aka termination codon).

The names of the `GENETIC_CODE` vector are the DNA codons i.e. the tri-nucleotide sequences (directed 5’ to 3’) that are assumed to belong to the "coding DNA strand" (aka "sense DNA strand" or "non-template DNA strand") of the gene.

The names of the `RNA_GENETIC_CODE` are the RNA codons i.e. the tri-nucleotide sequences (directed 5’ to 3’) that are assumed to belong to the mRNA of the gene.

Note that the values in the `GENETIC_CODE` and `RNA_GENETIC_CODE` vectors are the same, only their names are different. The names of the latter are those of the former where all occurrences of T (thymine) have been replaced by U (uracil).

**Author(s)**

H. Pages

**References**


**See Also**

`AA_ALPHABET, AMINO_ACID_CODE, trinucleotideFrequency, DNAString, RNAString, AAString`

**Examples**

```r
GENETIC_CODE RNA_GENETIC_CODE all(GENETIC_CODE == RNA_GENETIC_CODE) # TRUE
```

---

**IUPAC_CODE_MAP**

The IUPAC Extended Genetic Alphabet

**Description**

The `IUPAC_CODE_MAP` named character vector contains the mapping from the IUPAC nucleotide ambiguity codes to their meaning.

The `mergeIUPACLetters` function provides the reverse mapping.

**Usage**

```r
IUPAC_CODE_MAP
mergeIUPACLetters(x)
```

**Arguments**

- `x` A vector of non-empty character strings made of IUPAC letters.
Details

IUPAC nucleotide ambiguity codes are used for representing sequences of nucleotides where the exact nucleotides that occur at some given positions are not known with certainty.

Value

IUPAC_CODE_MAP is a named character vector where the names are the IUPAC nucleotide ambiguity codes and the values are their corresponding meanings. The meaning of each code is described by a string that enumerates the base letters ("A", "C", "G" or "T") associated with the code.

The value returned by mergeIUPACLetters is an unnamed character vector of the same length as its argument x where each element is an IUPAC nucleotide ambiguity code.

Author(s)

H. Pages

References

http://www.chick.manchester.ac.uk/SiteSeer/IUPAC_codes.html


See Also

DNAString, RNAString

Examples

IUPAC_CODE_MAP
some_iupac_codes <- c("R", "M", "G", "N", "V")
IUPAC_CODE_MAP[some_iupac_codes]
mergeIUPACLetters(IUPAC_CODE_MAP[some_iupac_codes])

class <- InDel

Description

The InDel class is a container for storing insertion and deletion information.

Details

This is a generic class that stores any insertion and deletion information.

Accesser methods

In the code snippets below, x is a InDel object.

insertion(x): The insertion information.
deletion(x): The deletion information.
The **MIndex class** is the basic container for storing the matches of a set of patterns in a subject sequence.

**Details**

**THIS IS STILL WORK IN PROGRESS!**

An MIndex object contains the matches (start/end locations) of a set of patterns found in an `XString` object called "the subject string" or "the subject sequence" or simply "the subject".

The `matchPDict` function returns an MIndex object.

MORE TO COME SOON...

### Accessor methods

In the code snippets below, `x` is an MIndex object.

- `length(x)`: The number of patterns that matches are stored for.
- `names(x)`: The names of the patterns that matches are stored for.
- `startIndex(x)`: A list containing the starting positions of the matches for each pattern.
- `endIndex(x)`: A list containing the ending positions of the matches for each pattern.
- `countIndex(x)`: An integer vector containing the number of matches for each pattern.

### Subsetting methods

In the code snippets below, `x` is an MIndex object.

- `x[[i]]`: Extract the matches for the i-th pattern as an `IRanges` object.

### Other utility methods and functions

In the code snippets below, `x` and `mindex` are MIndex objects and `subject` is the `XString` object containing the sequence in which the matches were found.

- `unlist(x, recursive=TRUE, use.names=TRUE)`: Return all the matches in a single `IRanges` object. `recursive` and `use.names` are ignored.
- `extractAllMatches(subject, mindex)`: Return all the matches in a single `XStringViews` object.
MaskedXString-class

Author(s)
H. Pages

See Also
matchPDict, PDict-class, IRanges-class, XStringViews-class

Examples

---

MaskedXString-class

MaskedXString objects

Description

The MaskedBString, MaskedDNAString, MaskedRNAString and MaskedAAString classes are containers for storing masked sequences.

All those containers derive directly (and with no additional slots) from the MaskedXString virtual class. They are also said to be MaskedXString subtypes.

Details

In Biostrings, a pile of masks can be put on top of a sequence. A pile of masks is represented by a MaskCollection object and the sequence by an XString object. A MaskedXString object is the result of bundling them together in a single object.

Note that, no matter what masks are put on top of it, the original sequence is always stored unmodified in a MaskedXString object. This allows the user to activate/deactivate masks without having to worry about losing the information stored in the masked/unmasked regions. Also this allows efficient memory management since the original sequence never needs to be copied (modifying it would require to make a copy of it first - sequences cannot and should never be modified in place in Biostrings), even when the set of active/inactive masks changes.

Accesor methods

In the code snippets below, x is a MaskedXString object. For masks(x) and masks(x) <- y, it can also be an XString object and y must be NULL or a MaskCollection object.

unmasked(x): Turns x into an XString object by dropping the masks.
masks(x): Turns x into a MaskCollection object by dropping the sequence.
masks(x) <- y: If x is an XString object and y is NULL, then this doesn’t do anything.
  If x is an XString object and y is a MaskCollection object, then this turns x into a MaskedXString object by putting the masks in y on top of it.
  If x is a MaskedXString object and y is NULL, then this is equivalent to x <- unmasked(x).
  If x is a MaskedXString object and y is a MaskCollection object, then this replaces the masks currently on top of x by the masks in y.
alphabet(x): Equivalent to alphabet(unmasked(x)). See ?alphabet for more information.
length(x): Equivalent to length(unmasked(x)). See ?length,XString-method` for more information.
"maskedwidth" and related methods

In the code snippets below, \( x \) is a MaskedXString object.

- \( \text{maskedwidth}(x) \): Get the number of masked letters in \( x \). A letter is considered masked iff it’s masked by at least one active mask.
- \( \text{maskedratio}(x) \): Equivalent to \( \text{maskedwidth}(x) / \text{length}(x) \).
- \( \text{nchar}(x) \): Equivalent to \( \text{length}(x) - \text{maskedwidth}(x) \).

Coercion

In the code snippets below, \( x \) is a MaskedXString object.

- \( \text{as}(x, \"XStringViews\") \): Turns \( x \) into an \( XStringViews \) object where the views are the unmasked regions of the original sequence ("unmasked" means not masked by at least one active mask).

Other methods

In the code snippets below, \( x \) is a MaskedXString object.

- \( \text{reduce}(x) \): Reduce the set of masks in \( x \) to a single mask made of all active masks.
- \( \text{gaps}(x) \): Reverses all the masks i.e. each mask is replaced by a mask where previously un-masked regions are now masked and previously masked regions are now un-masked.

Author(s)

H. Pages

See Also

- maskMotif
- injectHardMask
- alphabetFrequency
- reverse
- MaskedXString-method
- XString-class
- MaskCollection-class
- XStringViews-class
- IRanges-utils

Examples

```r
## A. MASKING BY POSITION
mask0 <- Mask(mask.width=29, start=c(3, 10, 25), width=c(6, 8, 5))
x <- DNAString("ACACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA\n\nACACTAGATA"")
length(x)  # same as width(mask0)
nchar(x)    # same as length(x)
masks(x) <- mask0
x
length(x)  # has not changed
nchar(x)    # has changed
gaps(x)

## Prepare a MaskCollection object of 3 masks (\'mymasks\') by running the
## examples in the man page for these objects:
example(MaskCollection, package="IRanges")

## Put it on \'x\':
masks(x) <- mymasks
```
PDict-class

## Deactivate all masks:
active(masks(x)) <- FALSE

## Activate mask "C":
active(masks(x))['C'] <- TRUE

## Turn MaskedXString object into an XStringViews object:
as(x, "XStringViews")

## Drop the masks:
masks(x) <- NULL

alphabetFrequency(x)

# ---------------------------------------------------------------------
# B. MASKING BY CONTENT
# # ---------------------------------------------------------------------
# # See ?maskMotif for masking by content

PDict-class  

PDict objects

Description

The PDict class is a container for storing a preprocessed dictionary of DNA patterns that can later be passed to the matchPDict function for fast matching.

PDict is the constructor function for creating new PDict objects.

Usage

PDict(x, max.mismatch=NA, tb.start=NA, tb.end=NA, tb.width=NA, type="ACtree", skip.invalidpatterns=FALSE)

Arguments

x  
A character vector, a DNAStringSet object or an XStringViews object with a DNAString subject.

max.mismatch  
A single non-negative integer or NA. See the "Allowing a small number of mismatching letters" section below.

tb.start  
A single integer or NA. See the "Trusted Band" section below.

tb.end  
A single integer or NA. See the "Trusted Band" section below.

tb.width  
A single integer or NA. See the "Trusted Band" section below.

type  
"ACtree" or "Twobit"

skip.invalidpatterns  
This argument is not supported yet (and might in fact be replaced by the filter argument very soon).
Details

THIS IS STILL WORK IN PROGRESS!

If the original dictionary \( x \) is a character vector or an XStringViews object with a DNAString subject, then the PDict constructor will first try to turn it into a DNAStringSet object.

By default (i.e. if PDict is called with max.mismatch=NA, tb.start=NA, tb.end=NA and tb.width=NA) the following limitations apply: (1) the original dictionary can only contain base letters (i.e. only As, Cs, Gs and Ts), therefore IUPAC extended letters are not allowed; (2) all the patterns in the dictionary must have the same length ("constant width" dictionary); and (3) later matchPdict can only be used with max.mismatch=0.

A Trusted Band can be used in order to relax these limitations (see the "Trusted Band" section below).

If you are planning to use the resulting PDict object in order to do inexact matching where valid hits are allowed to have a small number of mismatching letters, then see the "Allowing a small number of mismatching letters" section below.

Two types of preprocessing are currently supported: type="ACtree" (the default) and type="Twobit". With the "ACtree" type, all the oligonucleotides in the Trusted Band are stored in a 4-ary Aho-Corasick tree. With the "Twobit" type, the 2-bit-per-letter signatures of all the oligonucleotides in the Trusted Band are computed and the mapping from these signatures to the 1-based position of the corresponding oligonucleotide in the Trusted Band is stored in a way that allows very fast lookup. Only with PDict objects of type "ACtree" can matchPdict then be called with fixed="pattern" (instead of fixed=TRUE, the default) so that IUPAC extended letters in the subject are treated as ambiguities. PDict objects of type "Twobit" don’t allow this.

Trusted Band

What’s a Trusted Band?

A Trusted Band is a region defined in the original dictionary where the limitations described above will apply.

Why use a Trusted Band?

Because the limitations described above will apply to the Trusted Band only! For example the Trusted Band cannot contain IUPAC extended letters but the "head" and the "tail" can (see below for what those are). Also with a Trusted Band, if matchPdict is called with a non-null max.mismatch value then mismatching letters will be allowed in the head and the tail. Or, if matchPdict is called with fixed="subject", then IUPAC extended letters in the head and the tail will be treated as ambiguities.

How to specify a Trusted Band?

Use the tb.start, tb.end and tb.width arguments of the PDict constructor in order to specify a Trusted Band. This will divide each pattern in the original dictionary into three parts: a left part, a middle part and a right part. The middle part is defined by its starting and ending nucleotide positions given relatively to each pattern thru the tb.start, tb.end and tb.width arguments. It must have the same length for all patterns (this common length is called the width of the Trusted Band). The left and right parts are defined implicitly: they are the parts that remain before (prefix) and after (suffix) the middle part, respectively. Therefore three DNAStringSet objects result from this division: the first one is made of all the left parts and forms the head of the PDict object, the second one is made of all the middle parts and forms the Trusted Band of the PDict object, and the third one is made of all the right parts and forms the tail of the PDict object.

In other words you can think of the process of specifying a Trusted Band as drawing 2 vertical lines on the original dictionary (note that these 2 lines are not necessarily straight lines but the horizontal space between them must be constant). When doing this, you are dividing the dictionary
into three regions (from left to right): the head, the Trusted Band and the tail. Each of them is a DNAStringSet object with the same number of elements than the original dictionary and the original dictionary could easily be reconstructed from those three regions.

The width of the Trusted Band must be \( \geq 1 \) because Trusted Bands of width 0 are not supported.

Finally note that calling \texttt{PDict} with \texttt{tb.start=NA, tb.end=NA and tb.width=NA} (the default) is equivalent to calling it with \texttt{tb.start=1, tb.end=-1 and tb.width=NA}, which results in a full-width Trusted Band i.e. a Trusted Band that covers the entire dictionary (no head and no tail).

Allowing a small number of mismatching letters

\textbf{TODO}

\textbf{Accesor methods}

In the code snippets below, \( x \) is a \texttt{PDict} object.

\begin{itemize}
  \item \texttt{length(x)}: The number of patterns in \( x \).
  \item \texttt{width(x)}: A vector of non-negative integers containing the number of letters for each pattern in \( x \).
  \item \texttt{names(x)}: The names of the patterns in \( x \).
  \item \texttt{head(x)}: The head of \( x \) or \texttt{NULL} if \( x \) has no head.
  \item \texttt{tb(x)}: The Trusted Band defined on \( x \).
  \item \texttt{tb.width(x)}: The width of the Trusted Band defined on \( x \). Note that, unlike \texttt{width(tb(x))}, this is a single integer. And because the Trusted Band has a constant width, \texttt{tb.width(x)} is in fact equivalent to \texttt{unique(width(tb(x)))}, or to \texttt{width(tb(x))[1]}.
  \item \texttt{tail(x)}: The tail of \( x \) or \texttt{NULL} if \( x \) has no tail.
\end{itemize}

\textbf{Subsetting methods}

In the code snippets below, \( x \) is a \texttt{PDict} object.

\begin{itemize}
  \item \texttt{x[[i]]}: Extract the i-th pattern from \( x \) as a \texttt{DNAString} object.
\end{itemize}

\textbf{Other methods}

In the code snippet below, \( x \) is a \texttt{PDict} object.

\begin{itemize}
  \item \texttt{duplicated(x)}: [TODO]
  \item \texttt{patternFrequency(x)}: [TODO]
\end{itemize}

\textbf{Author(s)}

H. Pages

\textbf{References}

See Also

matchPDict, DNA_ALPHABET, DNAStringSet-class, XStringViews-class

Examples

```R
## A. NO HEAD AND NO TAIL (THE DEFAULT)
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe$sequence)
dict0
length(dict0) # The original dictionary.
unique(nchar(dict0)) # Patterns are 25-mers.
pdict0 <- PDict(dict0) # Store the original dictionary in
# a PDict object (preprocessing).
pdict0
class(pdict0)
length(pdict0) # Same as length(dict0).
tb.width(pdict0) # The width of the (implicit)
# Trusted Band.
sum(duplicated(pdict0))
table(patternFrequency(pdict0)) # 9 patterns are repeated 3 times.
pdict0[[1]]
pdict0[[5]]

## B. NO HEAD AND A TAIL
dict1 <- c("ACNG", "GT", "CGT", "AC")
pdict1 <- PDict(dict1, tb.end=2)
pdict1
class(pdict1)
length(pdict1)
width(pdict1)
head(pdict1)
tb(pdict1)
tb.width(pdict1)
width(tb(pdict1))
tail(pdict1)
pdict1[[3]]
```

Description

The `PairwiseAlignedFixedSubject` class is a container for storing an alignment. The `PairwiseAlignedFixedSubjectSummary` class is a container for storing the summary of an alignment.
Details

Before we define the notion of alignment, we introduce the notion of "filled-with-gaps subsequence". A "filled-with-gaps subsequence" of a string string1 is obtained by inserting 0 or any number of gaps in a subsequence of s1. For example L-A–ND and A–N-D are "filled-with-gaps subsequences" of LAND. An alignment between two strings string1 and string2 results in two strings (align1 and align2) that have the same length and are "filled-with-gaps subsequences" of string1 and string2.

For example, this is an alignment between LAND and LEAVES:

\[
\begin{align*}
L-A \\
LEA
\end{align*}
\]

An alignment can be seen as a compact representation of one set of basic operations that transforms string1 into align1. There are 3 different kinds of basic operations: "insertions" (gaps in align1), "deletions" (gaps in align2), "replacements". The above alignment represents the following basic operations:

- insert E at pos 2
- insert V at pos 4
- insert E at pos 5
- replace by S at pos 6 (N is replaced by S)
- delete at pos 7 (D is deleted)

Note that "insert X at pos i" means that all letters at a position \( \geq i \) are moved 1 place to the right before X is actually inserted.

There are many possible alignments between two given strings string1 and string2 and a common problem is to find the one (or those ones) with the highest score, i.e. with the lower total cost in terms of basic operations.

Accessor methods

In the code snippets below, \( x \) is a PairwiseAlignedFixedSubject object, except otherwise noted.

- \textbf{pattern(x)}: The AlignedXStringSet object for the pattern.
- \textbf{subject(x)}: The AlignedXStringSet object for the subject.
- \textbf{type(x)}: The type of the alignment ("global", "local", "overlap", "patternOverlap"). There is a method for PairwiseAlignedFixedSubjectSummary as well.
- \textbf{score(x)}: The score of the alignment (integer). There is a method for PairwiseAlignedFixedSubjectSummary as well.
- \textbf{nindel(x)}: An InDel object containing the number of insertions and deletions.
- \textbf{length(x)}: The length of the aligned(pattern(x)) and aligned(subject(x)). There is a method for PairwiseAlignedFixedSubjectSummary as well.
- \textbf{nchar(x)}: The nchar of the aligned(pattern(x)) and aligned(subject(x)). There is a method for PairwiseAlignedFixedSubjectSummary as well.
- \textbf{alphabet(x)}: Equivalent to alphabet(unaligned(subject(x))).
- \textbf{summary(object, ...)}: Generates a summary for the PairwiseAlignedFixedSubject.
- \textbf{aligned(x)}: Returns an XStringSet object containing the aligned patterns without insertions. This operation “aligns” the alignments.
as.character(x): Converts aligned(x) to a character vector.

as.matrix(x): Returns an "exploded" character matrix representation of aligned(x).

Views(subject, start=NA, end=NA, names=NULL): The XStringViews object that represents the pairwise alignments along unaligned(subject(subject)). The start and end arguments must be either NA or an integer vector of length 1 that denotes the offset from start(subject(subject)).

toString(x): Equivalent to toString(as.character(x)).

Subsetting methods

x[i]: Returns a new PairwiseAlignedFixedSubject object made of the selected elements.

rep(x, times): Returns a new PairwiseAlignedFixedSubject object made of the repeated elements.

Author(s)

P. Aboyoun and H. Pages

See Also

pairwiseAlignment, AlignedXStringSet-class, XString-class, XStringViews-class, match-utils

Examples

pattern <- AAStringSet(c("HLDNKLKTF", "HVDDMPNAL"))
subject <- AAString("SMDDTEKMSMKL")
nw1 <- pairwiseAlignment(pattern, subject, substitutionMatrix = "BLOSUM50", gapOpening = 0, gapExtension = 0)

pattern(nw1)
subject(nw1)
aligned(nw1)
as.character(nw1)
as.matrix(nw1)
nchar(nw1)
score(nw1)
nw1

QualityScaledXStringSet-class

QualityScaledBStringSet, QualityScaledDNAStringSet, QualityScaledRNAStringSet and QualityScaledAAStringSet objects

Description

The QualityScaledBStringSet class is a container for storing a BStringSet object with an XStringQuality object.

Similarly, the QualityScaledDNAStringSet (or QualityScaledRNAStringSet, or QualityScaledAAStringSet) class is a container for storing a DNAStringSet (or RNAStringSet, or AAStringSet) objects with an XStringQuality object.
Usage

## Constructors:

- `QualityScaledBStringSet(x, quality)`
- `QualityScaledDNAStringSet(x, quality)`
- `QualityScaledRNAStringSet(x, quality)`
- `QualityScaledAAStringSet(x, quality)`

Arguments

- **x**: Either a character vector, or an `XString`, `XStringSet` or `XStringViews` object.
- **quality**: An `XStringQuality` object.

Details

The `QualityScaledBStringSet`, `QualityScaledDNAStringSet`, `QualityScaledRNAStringSet` and `QualityScaledAAStringSet` functions are constructors that can be used to "naturally" turn `x` into an `QualityScaledXStringSet` object of the desired subtype.

Accessor methods

The `QualityScaledXStringSet` class derives from the `XStringSet` class hence all the accessor methods defined for an `XStringSet` object can also be used on an `QualityScaledXStringSet` object. Common methods include (in the code snippets below, `x` is an `QualityScaledXStringSet` object):

- `length(x)`: The number of sequences in `x`.
- `width(x)`: A vector of non-negative integers containing the number of letters for each element in `x`.
- `nchar(x)`: The same as `width(x)`.
- `names(x)`: NULL or a character vector of the same length as `x` containing a short user-provided description or comment for each element in `x`.
- `quality(x)`: The quality of the strings.

Subsetting and appending

In the code snippets below, `x` and `values` are `XStringSet` objects, and `i` should be an index specifying the elements to extract.

- `x[i]`: Return a new `QualityScaledXStringSet` object made of the selected elements.

Author(s)

P. Aboyoun

See Also

- `BStringSet-class`, `DNAStringSet-class`, `RNAStringSet-class`, `AAStringSet-class`, `XStringQuality-class`

Examples

```r
x1 <- DNAStringSet(c("TTGA", "CTCN"))
q1 <- PhredQuality(c("++,-", "6789"))
qx1 <- QualityScaledDNAStringSet(x1, q1)
qx1
```
RNAString-class

RNAString objects

Description
An RNAString object allows efficient storage and manipulation of a long RNA sequence.

Details
The RNAString class is a direct XString subtype (with no additional slot). Therefore all functions and methods described in the XString man page also work with an RNAString object (inheritance).

Unlike the BString container that allows storage of any single string (based on a single-byte character set) the RNAString container can only store a string based on the RNA alphabet (see below). In addition, the letters stored in an RNAString object are encoded in a way that optimizes fast search algorithms.

The RNA alphabet
This alphabet contains all letters from the IUPAC Extended Genetic Alphabet (see ?IUPAC_CODE_MAP) where "T" is replaced by "U" + the gap ("-"") and the hard masking ("+") letters. It is stored in the RNA_ALPHABET constant (character vector). The alphabet method also returns RNA_ALPHABET when applied to an RNAString object and is provided for convenience only.

Constructor-like functions and generics
In the code snippet below, x can be a single string (character vector of length 1), a BString object or a DNAString object.

RNAString(x, start=1, nchar=NA, check=TRUE): Tries to convert x into an RNAString object by reading nchar letters starting at position start in x.

Accessor methods
In the code snippet below, x is an RNAString object.

alphabet(x): If x is an RNAString object, then return the RNA alphabet (see above). See the corresponding man pages when x is a BString, DNAString or AAString object.

Author(s)
H. Pages

See Also
IUPAC_CODE_MAP, letter, XString-class, DNAString-class, reverseComplement, alphabetFrequency
Examples

RNA_BASES
RNA_ALPHABET
d <- DNAString("TTGAAA-CTC-N")
r <- RNAString(d)
r
alphabet(r)  # RNA_ALPHABET

## When comparing an RNASTring object with a DNAString object,
## U and T are considered equals:
r == d  # TRUE

XString-class

BString objects

Description

The BString class is a general container for storing a big string (a long sequence of characters) and
for making its manipulation easy and efficient.

The DNAString, RNAString and AAString classes are similar containers but with the more biology-
oriented purpose of storing a DNA sequence (DNAString), an RNA sequence (RNAString), or a
sequence of amino acids (AAString).

All those containers derive directly (and with no additional slots) from the XString virtual class.
They are also said to be XString subtypes.

Details

The 2 main differences between an XString object and a standard character vector are: (1) the data
stored in an XString object are not copied on object duplication and (2) an XString object can only
store a single string (see the XStringSet container for an efficient way to store a big collection of
strings in a single object).

Unlike the DNAString, RNAString and AAString containers that accept only a predefined set of
letters (the alphabet), a BString object can be used for storing any single string based on a single-byte character set.

Constructor-like functions and generics

In the code snippet below, x can be a single string (character vector of length 1) or an XString
object.

BString(x, start=1, nchar=NA, check=TRUE): Tries to convert x into a BString
object by reading nchar letters starting at position start in x.

Accessor methods

In the code snippets below, x is an XString object.

alphabet(x): NULL for a BString object. See the corresponding man pages when x is a
DNAString, RNAString or AAString object.

length(x) or nchar(x): Get the length of an XString object, i.e., its number of letters.
Coercion

In the code snippets below, x is an XString object.

as.character(x): Converts x to a character string.
toString(x): Equivalent to as.character(x).

Subsetting

In the code snippets below, x is an XString object.

x[i]: Return a new XString object made of the selected letters (subscript i must be an NA-free numeric vector specifying the positions of the letters to select). The returned object belongs to the same class (i.e. same XString subtype) as x.
Note that, unlike subseq, x[i] does copy the sequence data and therefore will be very inefficient for extracting a big number of letters (e.g. when i contains millions of positions).

Equality

In the code snippets below, e1 and e2 are XString objects.

e1 == e2: TRUE if e1 is equal to e2. FALSE otherwise.

Comparison between two XString objects of different subtypes (e.g. a BString object and a DNAString object) is not supported with one exception: a DNAString object and an RNAString object can be compared (see RNAString-class for more details about this).
Comparison between a BString object and a character string is also supported (see examples below).

e1 != e2: Equivalent to !(e1 == e2).

Author(s)

H. Pages

See Also

subseq, letter, DNAString-class, RNAString-class, AAString-class, XStringSet-class, XStringViews-class, reverse, XString-method

Examples

b <- BString("I am a BString object")
b
length(b)

## Extracting a linear subsequence
subseq(b)
subseq(b, start=3)
subseq(b, start=-3)
subseq(b, end=-3)
subseq(b, end=-3, width=5)

## Subsetting
b2 <- b[length(b):1]    # better done with reverse(b)

as.character(b2)
b2 == b        # FALSE
b2 == as.character(b2)  # TRUE

## b[1:length(b)] is equal but not identical to b!
b == b[1:length(b)]  # TRUE
identical(b, 1:length(b))  # FALSE

## This is because subsetting an XString object with [ makes a copy
## of part or all its sequence data. Hence, for the resulting object,
## the internal slot containing the memory address of the sequence
## data differs from the original. This is enough for identical() to
## see the 2 objects as different.

---

**XStringPartialMatches-class**

*XStringPartialMatches objects*

---

**Description**

WARNING: This class is currently under development and might not work properly! Full documentation will come later.

Please DO NOT TRY TO USE it for now. Thanks for your comprehension!

**Accessor methods**

In the code snippets below, `x` is an `XStringPartialMatches` object.

- `subpatterns(x)`: Not ready yet.
- `pattern(x)`: Not ready yet.

**Standard generic methods**

In the code snippets below, `x` is an `XStringPartialMatches` objects, and `i` can be a numeric or logical vector.

- `x[i]`: Return a new `XStringPartialMatches` object made of the selected views. `i` can be a numeric vector, a logical vector, NULL or missing. The returned object has the same subject as `x`.

**Author(s)**

H. Pages

**See Also**

`XStringViews-class`, `XString-class`, `letter`
XStringQuality-class

PhredQuality and SolexaQuality objects

Description

Objects for storing string quality measures.

Usage

### Constructors:

PhredQuality(x)

SolexaQuality(x)

Arguments

x

Either a character vector, BString, BStringSet, integer vector, or number vector of error probabilities.

Details

PhredQuality objects store characters that are interpreted as [0 - 99] quality measures by subtracting 33 from their ASCII decimal representation (e.g. ! = 0, ” = 1, # = 2, ...).

SolexaQuality objects store characters are interpreted as [-5 - 99] quality measures by subtracting 64 from their ASCII decimal representation (e.g. ; = -5, < = -4, = = -3, ...).

Author(s)

P. Aboyoun

See Also

pairwiseAlignment, PairwiseAlignedFixedSubject-class, DNAString-class, BStringSet-class

Examples

PhredQuality(0:40)

SolexaQuality(0:40)

PhredQuality(seq(1e-4,0.5,length=10))

SolexaQuality(seq(1e-4,0.5,length=10))
**XStringSet-class**

**BStringSet, DNAStringSet, RNAStringSet and AAStringSet objects**

**Description**

The BStringSet class is a container for storing a set of BString objects and for making its manipulation easy and efficient.

Similarly, the DNAStringSet (or RNAStringSet, or AAStringSet) class is a container for storing a set of DNAString (or RNAString, or AAString) objects.

All those containers derive directly (and with no additional slots) from the XStringSet virtual class. They are also said to be XStringSet subtypes.

**Usage**

```r
## Constructors:
BStringSet(x, start=NA, end=NA, width=NA, use.names=TRUE)
DNAStringSet(x, start=NA, end=NA, width=NA, use.names=TRUE)
RNAStringSet(x, start=NA, end=NA, width=NA, use.names=TRUE)
AAStringSet(x, start=NA, end=NA, width=NA, use.names=TRUE)
```

**Arguments**

- **x** Either a character vector, or an XString, XStringSet or XStringViews object.
- **start** Either NA, a single integer, or an integer vector of the same length as x specifying how x should be "narrowed" (see ?narrow for the details).
- **end** Either NA, a single integer, or an integer vector of the same length as x specifying how x should be "narrowed" (see ?narrow for the details).
- **width** Either NA, a single integer, or an integer vector of the same length as x specifying how x should be "narrowed" (see ?narrow for the details).
- **use.names** TRUE or FALSE. Should names be preserved?

**Details**

The BStringSet, DNAStringSet, RNAStringSet and AAStringSet functions are constructors that can be used to "naturally" turn x into an XStringSet object of the desired subtype.

They also allow the user to "narrow" the sequences contained in x via proper use of the start, end and/or width arguments. In this context, "narrowing" means dropping unwanted parts of x located at the beginning (prefix) or end (suffix) of each sequence in x.

The narrow function is a generic function (defined in the IRanges package) with a method for narrowing IRanges objects. Because XStringSet objects are a particular kind of IRanges objects (the XStringSet class is a subclass of the IRanges class), an XStringSet object y can be narrowed with narrow(y). Therefore the two following expressions are equivalent:

```r
DNAStringSet(x, start=s, end=e, width=w)
narrow(DNAStringSet(x), start=s, end=e, width=w)
```

but, besides being more convenient, the former is also more memory efficient on character vectors and would work even if the dropped parts contained letters that are not in the DNA alphabet (see ?DNA_ALPHABET).
Accesor methods

The XStringSet class derives from the IRanges class hence all the accessor methods defined for a IRanges object can also be used on an XStringSet object. In particular, the following methods are available (in the code snippets below, x is an XStringSet object:

`length(x)`: The number of sequences in x.

`width(x)`: A vector of non-negative integers containing the number of letters for each element in x.

`nchar(x)`: The same as width(x).

`names(x)`: NULL or a character vector of the same length as x containing a short user-provided description or comment for each element in x. These are the only data in an XStringSet object that can safely be changed by the user. All the other data are immutable! As a general recommendation, the user should never try to modify an object by accessing its slots directly.

Subsetting and appending

In the code snippets below, x and values are XStringSet objects, and i should be an index specifying the elements to extract.

`x[i]`: Return a new XStringSet object made of the selected elements.

`x[[i]]`: Extract the i-th XString object from x.

`append(x, values, after=length(x))`: Add sequences in values to x.

Other methods

In the code snippets below, x is an XStringSet object.

`as.character(x, use.names)`: Convert x to a character vector of the same length as x.

use.names controls whether or not names(x) should be used to set the names of the returned vector (default is TRUE).

`as.matrix(x, use.names)`: Return a character matrix containing the "exploded" representation of the strings. This can only be used on an XStringSet object with equal-width strings.

use.names controls whether or not names(x) should be used to set the row names of the returned matrix (default is TRUE).

`toString(x)`: Equivalent to toString(as.character(x)).

Ordering and related methods

In the code snippets below, x is an XStringSet object.

`order(x)`: Return a permutation which rearranges x into ascending or descending order.

`sort(x)`: Sort x into ascending order (equivalent to x[order(x)])..

Author(s)

H. Pages

See Also

BString-class, DNAString-class, RNAString-class, AAString-class, XStringViews-class, narrow, DNA_ALPHABET
Examples

```r
x0 <- c("#TTGA", "#-CTC-N")
x1 <- DNAStringSet(x0, start=2)
x1
names(x1)
names(x1)[2] <- "seqB"
x1

library(drosophila2probe)
x2 <- DNAStringSet(drosophila2probe$sequence)
x2

RNAStringSet(x2, start=2, end=-5) # does NOT copy the sequence data!
```

XStringSet-io  
Read/write an XStringSet or XStringViews object from/to a file

Description

Functions to read/write an XStringSet or XStringViews object from/to a file.

Usage

```r
## XStringSet object
read.BStringSet(file, format)
read.DNAStringSet(file, format)
read.RNAStringSet(file, format)
read.AAStringSet(file, format)
write.XStringSet(x, file="", format, width=80)

## XStringViews object
read.XStringViews(file, format, subjectClass, collapse="")
write.XStringViews(x, file="", format, width=80)

## Some related helper functions
FASTArecordsToCharacter(FASTArecs, use.names=TRUE)
CharacterToFASTArecords(x)
FASTArecordsToXStringViews(FASTArecs, subjectClass, collapse="")
XStringSetToFASTArecords(x)
```

Arguments

- `file`  
  Either a character string naming a file or a connection open for reading or writing. If "" (the default for `write.XStringSet` and `write.XStringViews`), then the functions write to the standard output connection (the console) unless redirected by `sink`.

- `format`  
  Only "fasta" is supported for now.

- `x`  
  For `write.XStringSet` and `write.XStringViews`, the object to write to file. For `CharacterToFASTArecords`, the (possibly named) character vector to be converted to a list of FASTA records as one returned by `readFASTA`. For `XStringSetToFASTArecords`, the `XStringSet` object to be converted to a list of FASTA records as one returned by `readFASTA`. 
width

subjectClass

collapse

FASTArecs

use.names

Details

Only FASTA files are supported for now.

Reading functions read.BStringSet, read.DNAStringSet, read.RNAStringSet, read.AAStringSet and read.XStringViews load sequences from a file into an XStringSet or XStringViews object.

Writing functions write.XStringSet and write.XStringViews write an XStringSet or XStringViews object to a file or connection.

FASTArecordsToCharacter, CharacterToFASTArecords, FASTArecordsToXStringViews and XStringSetToFASTArecords are helper functions used internally by write.XStringSet and read.XStringViews for switching between different representations of the same object.

See Also

fasta.info, readFASTA, writeFASTA, XStringSet-class, XStringViews-class, BString-class, DNAString-class, RNAString-class, AAString-class

Examples

defile <- system.file("extdata", "someORF.fa", package="Biostrings")
xx <- read.DNAStringSet(file, "fasta")

write.XStringSet(x, format="fasta") # writes to the console

# Converting 'x'
# ... to a list of FASTA records (as one returned by the "readFASTA" function)
x1 <- XStringSetToFASTArecords(x)
# ... to a named character vector
x2 <- FASTArecordsToCharacter(x1) # same as 'as.character(x)'
## Constructors:

### S4 method for signature 'character':

```
Views(subject, start=NA, end=NA, names=NULL)
```

### S4 method for signature 'XString':

```
Views(subject, start=NA, end=NA, names=NULL)
```

## Arguments

- **subject**: The subject sequence.
- **start, end**: Integer vectors specifying the starting and ending positions of each view.
- **names**: If not `NULL`, the names to assign to each view.

## Details

An `XStringViews` object contains a set of views (start/end locations) on the same `XString` object called "the subject string" or "the subject sequence" or simply "the subject". Each view is defined by its start and end locations: both are integers such that `start <= end`. An `XStringViews` object is in fact a particular case of an `Views` object (the `XStringViews` class contains the `Views` class) so it can be manipulated in a similar manner: see `?Views` for more information. Note that two views can overlap and that a view can be "out of limits" i.e. it can start before the first letter of the subject or end after its last letter.

## Accessor methods

In the code snippets below, `x` is an `XStringViews` object.

- **subject(x)**: The subject of `x`. This is always an `XString` object.
- **nchar(x)**: A vector of non-negative integers containing the number of letters in each view. Values in `nchar(x)` coincide with values in `width(x)` except for "out of limits" views where they are lower.

## Other methods

In the code snippets below, `x`, `object`, `e1` and `e2` are `XStringViews` objects, and `i` can be a numeric or logical vector.

- **e1 == e2**: A vector of logicals indicating the result of the view by view comparison. The views in the shorter of the two `XStringViews` object being compared are recycled as necessary.
  
  Like for comparison between `XString` objects, comparison between two `XStringViews` objects with subjects of different classes is not supported with one exception: when the subjects are `DNAString` and `RNAString` instances.
  
  Also, like with `XString` objects, comparison between an `XStringViews` object with a `BString` subject and a character vector is supported (see examples below).

- **e1 != e2**: Equivalent to `!(e1 == e2)`.

- **as.character(x, use.names, check.limits)**: Convert `x` to a character vector of the same length as `x`. `use.names` controls whether or not `names(x)` should be used to set the names of the returned vector (default is `TRUE`). `check.limits` controls whether or not an error should be raised if `x` contains "out of limit" views (default is `TRUE`). With `check.limits=FALSE` then "out of limit" views are padded with spaces.
as.matrix(x, mode, use.names, check.limits): Depending on what mode is chosen ("integer" or "character"), return either a 2-column integer matrix containing start(x) and end(x) or a character matrix containing the "exploded" representation of the views. mode="character" can only be used on an XStringViews object with equal-width views. Arguments use.names and check.limits are ignored with mode="integer". With mode="character", use.names controls whether or not names(x) should be used to set the row names of the returned matrix (default is TRUE), and check.limits controls whether or not an error should be raised if x contains "out of limit" views (default is TRUE). With check.limits=FALSE then "out of limit" views are padded with spaces.

toString(x): Equivalent to toString(as.character(x)).

Author(s)

H. Pages

See Also

Views-class, gaps, XStringViews-constructors, XString-class, XStringSet-class, letter, MIndex-class

Examples

## One standard way to create an XStringViews object is to use
## the Views() constructor.

## Views on a DNAString object:
s <- DNAString("-CTC-N")
v4 <- Views(s, start=3:0, end=5:8)
subject(v4)
length(v4)
start(v4)
end(v4)
width(v4)

## Attach a comment to views #3 and #4:
names(v4)[3:4] <- "out of limits"

## A more programatical way to "tag" the "out of limits" views:
names(v4)[start(v4) < 1 | nchar(subject(v4)) < end(v4)] <- "out of limits"

## Two equivalent ways to extract a view as an XString object:
s2a <- v4[2]
s2b <- subseq(subject(v4), start=start(v4)[2], end=end(v4)[2])
identical(s2a, s2b) # TRUE

## It is an error to try to extract an "out of limits" view:
#v4[3] # Error!

v12 <- Views(DNAString("TAATAATG"), start=-2:9, end=0:11)
v12 == DNAString("TAA")
v12[v12 == v12[4]]
v12[v12 == v12[1]]
Here the first view doesn't even overlap with the subject:
Views(BString("aaa--b"), start=-3:4, end=-3:4 + c(3:6, 6:3))

'start' and 'end' are recycled
Views("abcdefgхи", start=2:1, end=4)
Views("abcdefgхи", start=5:7)
Views("abcdefgхи", end=5:7)

Applying gaps() to an XStringViews object
v2 <- Views("abCDefgHIJK", start=c(8, 3), end=c(14, 4))
gaps(v2)

---

**XStringViews-constructors**

*Basic functions for creating or modifying XStringViews objects*

**Description**

A set of basic functions for creating or modifying XStringViews objects.

**Usage**

adjacentViews(subject, width, gapwidth=0)
XStringViews(x, subjectClass, collapse="")

**Arguments**

- **subject**: An XString object or a single string.
- **width**: An integer vector containing the widths of the views.
- **gapwidth**: An integer vector containing the widths of the gaps between the views.
- **x**: An XString object or a character vector for XStringViews. An XStringViews object for trim and subviews.
- **subjectClass**: The class to be given to the subject of the XStringViews object created and returned by the function. Must be the name of one of the direct XString subtypes i.e. "BString", "DNAString", "RNAString" or "AAString".
- **collapse**: An optional character string to be inserted between the views of the XStringViews object created and returned by the function.

**Details**

The adjacentViews function returns an XStringViews object containing views on subject with widths given in the width vector and separated by gaps of width gapwidth. The first view starts at position 1.

The XStringViews constructor will try to create an XStringViews object from the value passed to its x argument. If x itself is an XStringViews object, the returned object is obtained by coercing its subject to the class specified by subjectClass. If x is an XString object, the returned object is made of a single view that starts at the first letter and ends at the last letter of x (in addition x itself is coerced to the class specified by subjectClass when specified). If x is a character vector, the returned object has one view per character string in x (and its subject is an instance of the class specified by subjectClass).
Value

These functions return an XStringViews object y. length(y) (the number of views in y) is length(width) for the adjacentViews function. For the XStringViews constructor, length(y) is 1 when x is an XString object and length(x) otherwise.

See Also

XStringViews-class, XString-class

Examples

```r
adjacentViews("abcdefgij", 4:2, gapwidth=1)
v12 <- Views(DNAString("TAATAATG"), start=-2:9, end=0:11)
XStringViews(v12, subjectClass="RNAString")
XStringViews(AAString("MARKSLEMSIR*"))
XStringViews("abcdefgij", subjectClass="BString")
```

align-utils

Utility functions related to sequence alignment

Description

A variety of different functions used to deal with sequence alignments.

Usage

```r
mismatchTable(x, shiftLeft=0L, shiftRight=0L, ...)  
mismatchSummary(x, ...)  
# S4 method for signature 'AlignedXStringSet':  
coverage(x, start=NA, end=NA, weight=1L)  
# S4 method for signature 'PairwiseAlignedFixedSubject':  
coverage(x, start=NA, end=NA, weight=1L)  
compareStrings(pattern, subject)  
# S4 method for signature 'character':  
consensusMatrix(x, freq=FALSE)  
# S4 method for signature 'XStringSet':  
consensusMatrix(x, baseOnly=FALSE, freq=FALSE)  
consensusString(x)
```

Arguments

- `x`: A character vector or matrix, XStringSet, XStringViews, PairwiseAlignedFixedSubject or list of FASTA records containing the equal-length strings.
- `shiftLeft`, `shiftRight`: Non-positive and non-negative integers respectively that specify how many preceding and succeeding characters to and from the mismatch position to include in the mismatch substrings.
- `...`: Further arguments to be passed to or from other methods.
- `start`, `end`: See ?coverage.
weight  An integer vector specifying how much each element in \( x \) counts.

pattern, subject

The strings to compare. Can be of type character, XString, XStringSet, AlignedXStringSet, or, in the case of pattern, PairwiseAlignedFixedSubject. If pattern is a PairwiseAlignedFixedSubject object, then subject must be missing.

baseOnly  TRUE or FALSE. If TRUE, the returned vector only contains frequencies for the letters in the "base" alphabet i.e. "A", "C", "G", "T" if \( x \) is a "DNA input", and "A", "C", "G", "U" if \( x \) is an "RNA input". When \( x \) is a BString object (or an XStringViews object with a BString subject, or a BStringSet object), then the baseOnly argument is ignored.

freq  If TRUE, then letter frequencies (per position) are reported, otherwise counts.

Details

mismatchTable: a data.frame containing the positions and substrings of the mismatches for the AlignedXStringSet or PairwiseAlignedFixedSubject object.

mismatchSummary: a list of data.frame objects containing counts and frequencies of the mismatches for the AlignedXStringSet or PairwiseAlignedFixedSubject object.

compareStrings combines two equal-length strings that are assumed to be aligned into a single character string containing that replaces mismatches with "?", insertions with "+", and deletions with "-".

consensusMatrix computes a consensus matrix for a set of equal-length strings that are assumed to be aligned.

consensusString creates the string based on a 50% + 1 vote from the consensus matrix with unknowns labeled with "?".

See Also

pairwiseAlignment, XString-class, XStringSet-class, XStringViews-class, AlignedXStringSet-class, PairwiseAlignedFixedSubject-class, match-utils

Examples

```r
## Compare two globally aligned strings
string1 <- "ACTTCACAGCTTCCGTAGTGGAC---AAAGG---AACGCAAAGGTTTTCAG"
string2 <- "GTTTCACTACTTCCTTTCGGGTAAGTAAATATAAATAAATATTATTATTATTATTATTTTCAG"
compareStrings(string1, string2)

## Create a consensus matrix
nw1 <- pairwiseAlignment(AAStringSet(c("HLDNLKGT"), "HVDDMPNAL")), AAString("SMDDTEKMSKLM"),
    substitutionMatrix = "BLOSUM50", gapOpening = -3, gapExtension = -1)
consensusMatrix(nw1)

## Examine the consensus between the bacteriophage phi X174 genomes
data(phiX174Phage)
phageConsmat <- consensusMatrix(phiX174Phage, baseOnly = TRUE)
phageDiffs <- which(apply(phageConsmat, 2, max) < length(phiX174Phage))
phageDiffs
phageConsmat[,phageDiffs]

## Read in ORF data
file <- system.file("extdata", "someORF.fa", package="Biostrings")
orf <- read.DNAStringSet(file, "fasta")

## To illustrate, the following example assumes the ORF data
## to be aligned for the first 10 positions (patently false):
orf10 <- DNAStringSet(orf, end=10)
consensusMatrix(orf10, baseOnly=TRUE, freq=TRUE)
consensusString(sort(orf10)[1:5])

## For the character matrix containing the "exploded" representation
## of the strings, do:
as.matrix(orf10, use.names=FALSE)

---

**alphabetFrequency**  
*Function to calculate the frequency of letters in a biological sequence and related functions*

**Description**

Given a biological sequence, the *alphabetFrequency* function will calculate the frequency of each letter in the (base) alphabet, the *dinucleotideFrequency* function the frequency of all possible dinucleotides and the *trinucleotideFrequency* function the frequency of all possible trinucleotides.

More generally, the *oligonucleotideFrequency* function will calculate the frequency of all possible oligonucleotides of a given length (called the "width" in this particular context).

In this man page we call "DNA input" a DNAString object, or a DNAStringSet object, or an XStringViews object with a DNAString subject, or a MaskedDNAString object. Similarly we call "RNA input" an RNAString object, or an RNAStringSet object, or an XStringViews object with an RNAString subject, or a MaskedRNAString object.

**Usage**

```r
alphabetFrequency(x, baseOnly=FALSE, freq=FALSE, ...)
hasOnlyBaseLetters(x)
uniqueLetters(x)

dinucleotideFrequency(x, freq=FALSE, fast-moving.side="right", as.matrix=FALSE, with.labels=TRUE, ...)
trinucleotideFrequency(x, freq=FALSE, fast-moving.side="right", as.array=FALSE, with.labels=TRUE, ...)
oligonucleotideFrequency(x, width, freq=FALSE, fast-moving.side="right", as.array=FALSE, with.labels=TRUE, ...)
oligonucleotideTransitions(x, left=1, right=1, freq=FALSE)
```

## Some related utility functions

```r
strrev(x)
mkAllStrings(alphabet, width, fast-moving.side="right")
```
alphabetFrequency

Arguments

x  An XString, XStringSet, XStringViews or MaskedXString object for the +Frequency and uniqueLetters functions. "DNA or RNA input" for hasOnlyBaseLetters. A character vector for strrev.

baseOnly  TRUE or FALSE. If TRUE, the returned vector only contains frequencies for the letters in the "base" alphabet i.e. "A", "C", "G", "T" if x is a "DNA input", and "A", "C", "G", "U" if x is "RNA input". When x is a BString object (or an XStringViews object with a BString subject, or a BStringSet object), then the baseOnly argument is ignored.

collapse  If TRUE then frequencies are reported, otherwise counts.

freq  If TRUE then frequencies are reported, otherwise counts.

...  Further arguments to be passed to or from other methods. For the XStringViews and XStringSet methods, the collapse argument is accepted.

Details

alphabetFrequency and oligonucleotideFrequency are generic functions defined in the Biostrings package with methods defined for BString, DNAString, RNAString, XStringViews and XStringSet objects.

Value

All the +Frequency functions return an integer vector if freq is FALSE (default), otherwise a double vector. If as.matrix or as.array is TRUE, this vector is formatted as a matrix or an array.

For alphabetFrequency: if x is a "DNA or RNA input", then the returned vector is named with the letters in the alphabet (unless with.labels is FALSE). If the baseOnly argument is TRUE, then the returned vector has only 5 elements: 4 elements corresponding to the 4 nucleotides + the ‘other’ element.

dinucleotideFrequency (resp. trinucleotideFrequency and oligonucleotideFrequency) only works on "DNA or RNA input" and returns a vector named with all the possible dinucleotides (resp. trinucleotides or oligonucleotides).

If x is a multiple sequence input (i.e. an XStringViews or XStringSet object), then the returned object is a matrix (or a list) with the same number of rows (or elements) as x unless collapse=TRUE is specified. In that case the returned vector (or array) contains the frequencies cumulated across all sequences in x.
hasOnlyBaseLetters returns TRUE or FALSE indicating whether or not \( x \) contains only base letters (i.e. As, Cs, Gs and Ts for "DNA input" and As, Cs, Gs and Us for "RNA input").

uniqueLetters returns a vector of 1-letter or empty strings. The empty string is used to represent the nul character if \( x \) happens to contain any. Note that this can only happen if XString base subtype of \( x \) is BString.

**Author(s)**

H. Pages

**See Also**

countPDict, XString-class, XStringSet-class, XStringViews-class, MaskedXString-class, reverse, XString-method, rev, strsplit, GENETIC_CODE, AMINO_ACID_CODE

**Examples**

data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)

alphabetFrequency(yeast1)
alphabetFrequency(yeast1, baseOnly=TRUE)
hasOnlyBaseLetters(yeast1)
uniqueLetters(yeast1)

dinucleotideFrequency(yeast1)
trinucleotideFrequency(yeast1)
oligonucleotideFrequency(yeast1, 4)

## With a multiple sequence input
library(drosophila2probe)
x <- DNAStringSet(drosophila2probe$sequence)
alphabetFrequency(x[1:50], baseOnly=TRUE)
alphabetFrequency(x, baseOnly=TRUE, collapse=TRUE)

## Get the less and most represented 6-mers
f6 <- oligonucleotideFrequency(yeast1, 6)
f6[f6 == min(f6)]
f6[f6 == max(f6)]

## Get the result as an array
tri <- trinucleotideFrequency(yeast1, as.array=TRUE)
tri["A", "A", "C"] # == trinucleotideFrequency(yeast1)["AAC"]
tri["T", , ] # frequencies of trinucleotides starting with a "T"

## Get nucleotide transition matrices for yeast1
oligonucleotideTransitions(yeast1)
oligonucleotideTransitions(yeast1, 2, freq=TRUE)

## Note that when dropping the dimensions of the 'tri' array, elements
## in the resulting vector are ordered as if they were obtained with
## "fast.moving.side="left"":
triL <- trinucleotideFrequency(yeast1, fast.moving.side="left")
all(as.vector(tri) == triL) # TRUE

## Convert the trinucleotide frequency into the amino acid frequency based on
## translation

```
tri1 <- trinucleotideFrequency(yeast1)
names(tri1) <- GENETIC_CODE[names(tri1)]
sapply(split(tri1, names(tri1)), sum) # 12512 occurrences of the stop codon
```

## When the returned vector is very long (e.g. width >= 10), using 'with.labels=FALSE' will improve the performance considerably (100x, 1000x or more):
```
f12 <- oligonucleotideFrequency(yeast1, 12, with.labels=FALSE) # very fast!
```

## Some related utility functions
```
dict1 <- mkAllStrings(LETTERS[1:3], 4)
dict2 <- mkAllStrings(LETTERS[1:3], 4, fast.moving.side="left")
identical(strrev(dict1), dict2) # TRUE
```

---

**chartr**  
*Translating letters of a sequence*

**Description**

Translate letters of a sequence.

**Usage**

```
chartr(old, new, x)
```

**Arguments**

- `old`: A character string specifying the characters to be translated.
- `new`: A character string specifying the translations.
- `x`: The sequence or set of sequences to translate. If `x` is an `XString`, `XStringSet`, `XStringViews` or `MaskedXString` object, then the appropriate `chartr` method is called, otherwise the standard `chartr` R function is called.

**Details**

See `?chartr` for the details.

Note that, unlike the standard `chartr` R function, the methods for `XString`, `XStringSet`, `XStringViews` and `MaskedXString` objects do NOT support character ranges in the specifications.

**Value**

An object of the same class and length as the original object.

**See Also**

`chartr`, `replaceLetterAt`, `XString-class`, `XStringSet-class`, `XStringViews-class`, `MaskedXString-class`, `alphabetFrequency`, `matchPattern`, `reverseComplement`
Examples

```r
x <- BString("MiXeD cAsE 123")
chartr("IXs", "why", x)
```

```r
library(BSgenome.Celegans.UCSC.ce2)
chrII <- Celegans["chrII"]
alphabetFrequency(chrII)
pattern <- DNAString("TGGGTGTATTTA")
```

```r
## Transforming and searching the + strand
plus_strand <- chartr("C", "T", chrII)
alphabetFrequency(plus_strand)
matchPattern(pattern, plus_strand)
matchPattern(pattern, chrII)
```

```r
## Transforming and searching the - strand
minus_strand <- chartr("G", "A", chrII)
alphabetFrequency(minus_strand)
matchPattern(reverseComplement(pattern), minus_strand)
matchPattern(reverseComplement(pattern), chrII)
```

---

findPalindromes

*Searching a sequence for palindromes or complemented palindromes*

Description

The `findPalindromes` and `findComplementedPalindromes` functions can be used to find palindromic or complemented palindromic regions in a sequence.

`palindromeArmLength`, `palindromeLeftArm`, `palindromeRightArm`, `complementedPalindromeArmLength`, `complementedPalindromeLeftArm` and `complementedPalindromeRightArm` are utility functions for operating on palindromic or complemented palindromic sequences.

Usage

```r
findPalindromes(subject, min.armlength=4, max.looplength=1, min.looplength=0, palindromeArmLength(x, max.mismatch=0, ...)
palindromeLeftArm(x, max.mismatch=0, ...)
palindromeRightArm(x, max.mismatch=0, ...)
```

```r
findComplementedPalindromes(subject, min.armlength=4, max.looplength=1, min.looplength=0, complementedPalindromeArmLength(x, max.mismatch=0, ...)
complementedPalindromeLeftArm(x, max.mismatch=0, ...)
complementedPalindromeRightArm(x, max.mismatch=0, ...)
```

Arguments

- **subject**: An `XString` object containing the subject string, or an `XStringViews` object.
**findPalindromes**

- **min.armLength**
  An integer giving the minimum length of the arms of the palindromes (or complemented palindromes) to search for.

- **max.loopLength**
  An integer giving the maximum length of "the loop" (i.e. the sequence separating the 2 arms) of the palindromes (or complemented palindromes) to search for. Note that by default (max.loopLength=1), findPalindromes will search for strict palindromes (or complemented palindromes) only.

- **min.loopLength**
  An integer giving the minimum length of "the loop" of the palindromes (or complemented palindromes) to search for.

- **max.mismatch**
  The maximum number of mismatching letters allowed between the 2 arms of the palindromes (or complemented palindromes) to search for.

- **x**
  An XString object containing a 2-arm palindrome or complemented palindrome, or an XStringViews object containing a set of 2-arm palindromes or complemented palindromes.

- **...**
  Additional arguments to be passed to or from methods.

**Details**

The findPalindromes function finds palindromic substrings in a subject string. The palindromes that can be searched for are either strict palindromes or 2-arm palindromes (the former being a particular case of the latter) i.e. palindromes where the 2 arms are separated by an arbitrary sequence called "the loop".

Use the findComplementedPalindromes function to find complemented palindromic substrings in a DNAString subject (in a complemented palindrome the 2 arms are reverse-complementary sequences).

**Value**

findPalindromes and findComplementedPalindromes return an XStringViews object containing all palindromes (or complemented palindromes) found in subject (one view per palindromic substring found).

palindromeArmLength and complementedPalindromeArmLength return the arm length (integer) of the 2-arm palindrome (or complemented palindrome) x. It will raise an error if x has no arms. Note that any sequence could be considered a 2-arm palindrome if we were OK with arms of length 0 but we are not: x must have arms of length greater or equal to 1 in order to be considered a 2-arm palindrome. The same apply to 2-arm complemented palindromes. When applied to an XStringViews object x, palindromeArmLength and complementedPalindromeArmLength behave in a vectorized fashion by returning an integer vector of the same length as x.

palindromeLeftArm and complementedPalindromeLeftArm return an object of the same class as the original object x and containing the left arm of x.

palindromeRightArm does the same as palindromeLeftArm but on the right arm of x. Like palindromeArmLength, both palindromeLeftArm and palindromeRightArm will raise an error if x has no arms. Also, when applied to an XStringViews object x, both behave in a vectorized fashion by returning an XStringViews object of the same length as x.

**Author(s)**

H. Pages


## Description

This is a replacement for the standard gregexpr function that does exact matching only. Standard gregexpr() misses matches when they are overlapping. The gregexpr2 function finds all matches but it only works in "fixed" mode i.e. for exact matching (regular expressions are not supported).

## Usage

```r
gregexpr2(pattern, text)
```

## Arguments

- `pattern` character string to be matched in the given character vector
- `text` a character vector where matches are sought

## Value

A list of the same length as `text` each element of which is an integer vector as in gregexpr, except that the starting positions of all (even overlapping) matches are given. Note that, unlike gregexpr, gregexpr2 doesn’t attach a "match.length" attribute to each element of the returned list because, since it only works in "fixed" mode, then all the matches have the length of the pattern. Another difference with gregexpr is that with gregexpr2, the `pattern` argument must be a single (non-NA, non-empty) string.
**injektHardMask**

**Author(s)**

H. Pages

**See Also**

`gregexpr`, `matchPattern`

**Examples**

```r
gregexpr("aa", c("XaaaYaa", "a"), fixed=TRUE)
gregexpr2("aa", c("XaaaYaa", "a"))
```

---

**Description**

`injektHardMask` allows the user to "fill" the masked regions of a sequence with an arbitrary letter (typically the "+" letter).

**Usage**

```r
injektHardMask(x, letter="+")
```

**Arguments**

- `x` A `MaskedXString` or `XStringViews` object.
- `letter` A single letter.

**Details**

The name of the `injektHardMask` function was chosen because of the primary use that it is intended for: converting a pile of active "soft masks" into a "hard mask". Here the pile of active "soft masks" refers to the active masks that have been put on top of a sequence. In Biostrings, the original sequence and the masks defined on top of it are bundled together in one of the dedicated containers for this: the `MaskedBString`, `MaskedDNAString`, `MaskedRNAString` and `MaskedAASString` containers (this is the `MaskedXString` family of containers). The original sequence is always stored unmodified in a `MaskedXString` object so no information is lost. This allows the user to activate/deactivate masks without having to worry about losing the letters that are in the regions that are masked/unmasked. Also this allows better memory management since the original sequence never needs to be copied, even when the set of active/inactive masks changes.

However, there are situations where the user might want to really get rid of the letters that are in some particular regions by replacing them with a junk letter (e.g. "+") that is guaranteed to not interfere with the analysis that s/he is currently doing. For example, it’s very likely that a set of motifs or short reads will not contain the "+" letter (this could easily be checked) so they will never hit the regions filled with "+". In a way, it’s like the regions filled with "+" were masked but we call this kind of masking "hard masking".

Some important differences between "soft" and "hard" masking:

- `injektHardMask` creates a (modified) copy of the original sequence. Using "soft masking" does not.
A function that is "mask aware" like `alphabetFrequency` or `matchPattern` will really skip the masked regions when "soft masking" is used i.e. they will not walk thru the regions that are under active masks. This might lead to some speed improvements when a high percentage of the original sequence is masked. With "hard masking", the entire sequence is walked thru. Matches cannot span over masked regions with "soft masking". With "hard masking" they can.

Value

An `XString` object of the same length as the original object `x` if `x` is a `MaskedXString` object, or of the same length as `subject(x)` if it’s an `XStringViews` object.

Author(s)

H. Pages

See Also

`maskMotif`, `MaskedXString-class`, `replaceLetterAt`, `chartr`, `XString`, `XStringViews-class`

Examples

```r
## A. WITH AN XStringViews OBJECT
v2 <- Views("abCDefgHIJK", start=c(8, 3), end=c(14, 4))
injectHardMask(v2)
injectHardMask(v2, letter="=")

## B. WITH A MaskedXString OBJECT
mask0 <- Mask(mask.width=29, start=c(3, 10, 25), width=c(6, 8, 5))
x <- DNAString("ACACAATAGATAGNAGAGAGACG")
masks(x) <- mask0
subject <- injectHardMask(x)

## Matches can span over masked regions with "hard masking":
matchPattern("ACggggggA", subject, max.mismatch=6)
## but not with "soft masking":
matchPattern("ACggggggA", x, max.mismatch=6)
```

letter

Subsetting a string

Description

Extract a substring from a string by picking up individual letters by their position.

Usage

```r
letter(x, i)
```
maskMotif

Arguments

x  A character vector, or an XString, XStringViews or MaskedXString object.
i  An integer vector with no NAs.

Details

Unlike with the `substr` or `substring` functions, `i` must contain valid positions.

Value

A character vector of length 1 when `x` is an XString or MaskedXString object (the masks are ignored for the latter).

A character vector of the same length as `x` when `x` is a character vector or an XStringViews object.

Note that, because `i` must contain valid positions, all non-NA elements in the result are guaranteed to have exactly `length(i)` characters.

See Also

`subseq`, XString-class, XStringViews-class, MaskedXString-class

Examples

```r
x <- c("abcd", "ABC")
i <- c(3, 1, 1, 2, 1)

## With a character vector
letter(x[1], 3:1)
letter(x, 3)
letter(x, i)
#letter(x, 4)  # Error!

## With a BString object
letter(BString(x[1]), i)  # character vector
BString(x[1])[i]  # BString object

## With an XStringViews object
x2 <- XStringViews(x, "BString")
letter(x2, i)
```

maskMotif  

Masking by content (or by position)

Description

Functions for masking a sequence by content (or by position).

Usage

```r
maskMotif(x, motif, min.block.width=1)
mask(x, start=NA, end=NA, pattern)
```
Arguments

- **x**: The sequence to mask.
- **motif**: The motif to mask in the sequence.
- **min.block.width**: The minimum width of the blocks to mask.
- **start**: An integer vector containing the starting positions of the regions to mask.
- **end**: An integer vector containing the ending positions of the regions to mask.
- **pattern**: The motif to mask in the sequence.

Value

A **MaskedXString** object for `maskMotif` and an **XStringViews** object for `mask`.

Author(s)

H. Pages

See Also

`read.Mask`, **XString-class**, **MaskedXString-class**, **XStringViews-class**, **MaskCollection-class**

Examples

```r
## EXAMPLE 1
##
maskMotif(BString("AbcbcbEEEEE"), "bcb")
maskMotif(BString("AbcbcbEEEEE"), "bcb")

## maskMotif() can be used in an incremental way to mask more than 1
## motif. Note that maskMotif() does not try to mask again what's
## already masked (i.e. the new mask will never overlaps with the
## previous masks) so the order in which the motifs are masked actually
## matters as it will affect the total set of masked positions.
x0 <- BString("AbcbEEEcbcbEcbccbc")
x1 <- maskMotif(x0, "E")
x1x2 <- maskMotif(x1, "bcb")
x2x3 <- maskMotif(x2, "b")
x3

## Note that inverting the order in which "b" and "bcb" are masked would
## lead to a different final set of masked positions.
## Also note that the order doesn't matter if the motifs to mask don't
## overlap (we assume that the motifs are unique) i.e. if the prefix of
## each motif is not the suffix of any other motif. This is of course
## the case when all the motifs have only 1 letter.

## EXAMPLE 2
##
x <- DNAString("ACACAACTAGATAGNACTNNGAGAGACGC")
```
## Mask the N-blocks

```r
x1 <- maskMotif(x, "N")
x1
as(x1, "XStringViews")
gaps(x1)
as(gaps(x1), "XStringViews")
```

## Mask the AC-blocks

```r
x2 <- maskMotif(x1, "AC")
x2
gaps(x2)
```

## Mask the GA-blocks

```r
x3 <- maskMotif(x2, "GA", min.block.width=5)
x3 # masks 2 and 3 overlap
gaps(x3)
```

## EXAMPLE 3

```r
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrU <- Dmelanogaster$chrU
chrU
alphabetFrequency(chrU)
```

```r
chrU <- maskMotif(chrU, "N")
chrU
alphabetFrequency(chrU)
```

```r
as(chrU, "XStringViews")
as(gaps(chrU), "XStringViews")
```

```r
mask2 <- Mask(mask.width=length(chrU), start=c(50000, 350000, 543900), width=25000)
names(mask2) <- "some ugly regions"
masks(chrU) <- append(masks(chrU), mask2)
chrU
as(chrU, "XStringViews")
as(gaps(chrU), "XStringViews")
```

## EXAMPLE 4

```r
## Note that unlike maskMotif(), mask() returns an XStringViews object!

## masking "by position"
mask("AxyxyxBC", 2, 6)
```

```r
## masking "by content"
mask("AxyxyxBC", "xyx")
```

```r
noN_chrU <- mask(chrU, "N")
noN_chrU
alphabetFrequency(noN_chrU, collapse=TRUE)
```
match-utils

Description

In this man page we define precisely and illustrate what a "match" of a pattern P in a subject S is in the context of the Biostrings package. This definition of a "match" is central to most pattern matching functions available in this package: unless specified otherwise, most of them will adhere to the definition provided here.

neditStartingAt, neditEndingAt, isMatchingStartingAt and isMatchingEndingAt are low-level functions that implement some basic concepts. Once these concepts are understood, we can use them to provide a simple and concise definition of a "match".

Other utility functions related to pattern matching are described here: the mismatch function for getting the positions of the mismatching letters of a given pattern relatively to its matches in a given subject, the nmatch and nmismatch functions for getting the number of matching and mismatching letters produced by the mismatch function, and the coverage function that can be used to get the "coverage" of a subject by a given pattern or set of patterns.

Usage

neditStartingAt(pattern, subject, starting.at=1, with.indels=FALSE, fixed=TRUE)
neditEndingAt(pattern, subject, ending.at=1, with.indels=FALSE, fixed=TRUE)
neditAt(pattern, subject, at=1, with.indels=FALSE, fixed=TRUE)

isMatchingStartingAt(pattern, subject, starting.at=1, max.mismatch=0, with.indels=FALSE, fixed=TRUE)
isMatchingEndingAt(pattern, subject, ending.at=1, max.mismatch=0, with.indels=FALSE, fixed=TRUE)
isMatchingAt(pattern, subject, at=1, max.mismatch=0, with.indels=FALSE, fixed=TRUE)

mismatch(pattern, x, fixed=TRUE)
nmatch(pattern, x, fixed=TRUE)
nmismatch(pattern, x, fixed=TRUE)

## S4 method for signature 'MIndex':
coverage(x, start=NA, end=NA)

## S4 method for signature 'XStringViews':
coverage(x, start=NA, end=NA, weight=1L)

## S4 method for signature 'MaskedXString':
coverage(x, start=NA, end=NA, weight=1L)

Arguments

pattern

The pattern string.

subject

An XString object (or character vector) containing the subject sequence.

starting.at, ending.at, at

An integer vector specifying the starting (for starting.at and at) or ending (for ending.at) positions of the pattern relatively to the subject.

max.mismatch

See details below.

with.indels

See details below.

fixed

Only with a DNAString or RNAString subject can a fixed value other than the default (TRUE) be used.

With fixed=FALSE, ambiguities (i.e. letters from the IUPAC Extended Genetic Alphabet (see IUPAC_CODE_MAP) that are not from the base alphabet)
in the pattern \texttt{\_and\_} in the subject are interpreted as wildcards i.e. they match any letter that they stand for.

\texttt{fixed} can also be a character vector, a subset of \texttt{c(\"pattern\", \"subject\")}. \texttt{fixed=c(\"pattern\", \"subject\")} is equivalent to \texttt{fixed=TRUE} (the default). An empty vector is equivalent to \texttt{fixed=FALSE}. With \texttt{fixed=\"subject\"}, ambiguities in the pattern only are interpreted as wildcards. With \texttt{fixed=\"pattern\"}, ambiguities in the subject only are interpreted as wildcards.

\texttt{x} 
An \texttt{XStringViews} object for \texttt{mismatch} (typically, one returned by \texttt{matchPattern(pattern, subject)}).

Typically an \texttt{XStringViews} or \texttt{MIndex} object for \texttt{coverage} but \texttt{IRanges, MaskCollection} and \texttt{MaskedXString} objects are accepted too.

\texttt{start, end} 
Two single integers specifying where to start and end the extraction of the coverage in \texttt{x}.

\texttt{weight} 
An integer vector specifying how much each element in \texttt{x} counts.

\section*{Details}

A "match" of pattern \texttt{P} in subject \texttt{S} is a substring \texttt{S'} of \texttt{S} that is considered similar enough to \texttt{P} according to some distance (or metric) specified by the user. 2 distances are supported by most pattern matching functions in the Biostrings package. The first (and simplest) one is the "number of mismatching letters". It is defined only when the 2 strings to compare have the same length, so when this distance is used, only matches that have the same number of letters as \texttt{P} are considered. The second one is the "edit distance" (aka Levenshtein distance): it's the minimum number of operations needed to transform \texttt{P} into \texttt{S'}, where an operation is an insertion, deletion, or substitution of a single letter. When this metric is used, matches can have a different number of letters than \texttt{P}.

The \texttt{neditStartingAt} (and \texttt{neditEndingAt}) function implements these 2 distances. If \texttt{with.indels} is \texttt{FALSE} (the default), then the first distance is used i.e. \texttt{neditStartingAt} returns the "number of mismatching letters" between the pattern \texttt{P} and the substring \texttt{S'} of \texttt{S} starting at the positions specified in \texttt{starting.at} (note that \texttt{neditStartingAt} and \texttt{neditEndingAt} are vectorized so long vectors of integers can be passed thru the \texttt{starting.at} or \texttt{ending.at} arguments). If \texttt{with.indels} is \texttt{TRUE}, then the "edit distance" distance is used: for each position specified in \texttt{starting.at}, \texttt{P} is compared to all the substrings \texttt{S'} of \texttt{S} starting at this position and the smallest distance is returned. Note that this distance is guaranteed to be reached for a substrings of length < 2*length(\texttt{P}) so, of course, in practise, \texttt{P} only needs to be compared to a small number of substrings for every starting position.

\section*{Value}

\texttt{neditStartingAt} and \texttt{neditEndingAt}: an integer vector of the same length as \texttt{starting.at} (or \texttt{ending.at}).

\texttt{isMatchingStartingAt(\ldots)} and \texttt{isMatchingEndingAt(\ldots)}: the logical vector defined by \texttt{neditStartingAt(\ldots) <= max.mismatch} or \texttt{neditEndingAt(\ldots) <= max.mismatch}, respectively.

\texttt{neditAt} and \texttt{isMatchingAt} are convenience wrappers for \texttt{neditStartingAt} and \texttt{isMatchingStartingAt} respectively.

\texttt{mismatch}: a list of integer vectors.

\texttt{nmismatch}: an integer vector containing the length of the vectors produced by \texttt{mismatch}.

\texttt{coverage}: an \texttt{XRleInteger} object indicating the coverage of \texttt{x} in the interval specified by the \texttt{start} and \texttt{end} arguments. An integer value called the "coverage" can be associated to each position in \texttt{x}, indicating how many times this position is covered by the views or matches stored in
x. For example, if x is an XStringViews object, the coverage of a given position in x is the number of views it belongs to. If x is an MIndex object, the coverage of a given position in x is the number of matches (or hits) it belongs to. Note that the positions in the returned XRleInteger object are to be interpreted as relative to the interval specified by the start and end arguments.

See Also

matchPattern, matchPDict, IUPAC_CODE_MAP, XString-class, XStringViews-class, MIndex-class, coverage, IRanges-class, MaskCollection-class, MaskedXString-class, align-utils

Examples

```r
## ---------------------------------------------------------------------
## neditAt() / isMatchingAt()
## ---------------------------------------------------------------------
subject <- DNAString("GTATA")

## Pattern "AT" matches subject "GTATA" at position 3 (exact match)
neditAt("AT", subject, at=3)
isMatchingAt("AT", subject, at=3)

## ... but not at position 1
neditAt("AT", subject)
isMatchingAt("AT", subject)

## ... unless we allow 1 mismatching letter (inexact match)
isMatchingAt("AT", subject, max.mismatch=1)

## Here we look at 6 different starting positions and find 3 matches if
## we allow 1 mismatching letter
isMatchingAt("AT", subject, at=0:5, max.mismatch=1)

## No match
neditAt("NT", subject, at=1:4)
isMatchingAt("NT", subject, at=1:4)

## 2 matches if N is interpreted as an ambiguity (fixed=FALSE)
neditAt("NT", subject, at=1:4, fixed=FALSE)
isMatchingAt("NT", subject, at=1:4, fixed=FALSE)

## max.mismatch != 0 and fixed=FALSE can be used together
neditAt("NCA", subject, at=0:5, fixed=FALSE)
isMatchingAt("NCA", subject, at=0:5, max.mismatch=1, fixed=FALSE)

some_starts <- c(10:-10, NA, 6)
subject <- DNAString("ACGTGCA")
is_matching <- isMatchingAt("CAT", subject, at=some_starts, max.mismatch=1)
some_starts[is_matching]

## ---------------------------------------------------------------------
## mismatch() / nmismatch()
## ---------------------------------------------------------------------
m <- matchPattern("NCA", subject, max.mismatch=1, fixed=FALSE)
mismatch("NCA", m)
nmismatch("NCA", m)

## ---------------------------------------------------------------------
```
### Description

The `matchLRPatterns` function finds paired matches in a sequence i.e. matches specified by a left pattern, a right pattern and a maximum distance between the left pattern and the right pattern.

### Usage

```r
matchLRPatterns(Lpattern, Rpattern, max.ngaps, subject, 
    max.Lmismatch=0, max.Rmismatch=0, 
    Lfixed=TRUE, Rfixed=TRUE)
```

### Arguments

- **Lpattern**: The left part of the pattern.
- **Rpattern**: The right part of the pattern.
- **max.ngaps**: The max number of gaps in the middle i.e the max distance between the left and right parts of the pattern.
- **subject**: An XString, XStringViews or MaskedXString object containing the target sequence.
- **max.Lmismatch**: The maximum number of mismatching letters allowed in the left part of the pattern. If non-zero, an inexact matching algorithm is used (see the `matchPattern` function for more information).
- **max.Rmismatch**: Same as max.Lmismatch but for the right part of the pattern.
- **Lfixed**: Only with a DNAString or RNAString subject can a Lfixed value other than the default (TRUE) be used. With Lfixed=FALSE, ambiguities (i.e. letters from the IUPAC Extended Genetic Alphabet (see `IUPAC_CODE_MAP`) that are not from the base alphabet) in the left pattern _and_ in the subject are interpreted as wildcards i.e. they match any letter that they stand for. See the fixed argument of the `matchPattern` function for more information.
- **Rfixed**: Same as Lfixed but for the right part of the pattern.

### Value

An XStringViews object containing all the matches, even when they are overlapping (see the examples below), and where the matches are ordered from left to right (i.e. by ascending starting position).
matchPDict

Searching a sequence for patterns stored in a preprocessed dictionary

Description

The `matchPDict`, `countPDict` and `whichPDict` functions efficiently find the occurrences in a text (the subject) of all patterns stored in a preprocessed dictionary.

The three functions differ in what they return: `matchPDict` returns the "where" information i.e. the positions in the subject of all the occurrences of every pattern; `countPDict` returns the "how many times" information i.e. the number of occurrences for each pattern; and `whichPDict` returns the "who" information i.e. which patterns in the preprocessed dictionary have at least one match.

This man page shows how to use `matchPDict/countPDict/whichPDict` for exact matching of a constant width dictionary i.e. a dictionary where all the patterns have the same length (same number of nucleotides).

See `?matchPDict-inexact` for how to use these functions for inexact matching or when the original dictionary has a variable width.

Usage

```r
matchPDict(pdict, subject, algorithm="auto",
           max.mismatch=0, fixed=TRUE, verbose=FALSE)
countPDict(pdict, subject, algorithm="auto",
           max.mismatch=0, fixed=TRUE, verbose=FALSE)
whichPDict(pdict, subject, algorithm="auto",
           max.mismatch=0, fixed=TRUE, verbose=FALSE)
```

Examples

```r
library(BSgenome.Dmelanogaster.UCSC.dm3)
subject <- Dmelanogaster$chr3R
Lpattern <- "AGCTCCGAG"
Rpattern <- "TTGTTCACA"
matchLRPatterns(Lpattern, Rpattern, 500, subject) # 1 match

## Note that matchLRPatterns() will return all matches, even when they are
## overlapping:
subject <- DNAString("AAATTAACCCTT")
matchLRPatterns("AA", "TT", 0, subject) # 1 match
matchLRPatterns("AA", "TT", 1, subject) # 2 matches
matchLRPatterns("AA", "TT", 3, subject) # 3 matches
matchLRPatterns("AA", "TT", 7, subject) # 4 matches
```
**matchPDict**

Arguments

- **pdict**: A PDict object containing the preprocessed dictionary.
- **subject**: An XString object containing the subject string. For now, only XString subjects of subtype DNAString are supported.
- **algorithm**: Not supported yet.
- **max.mismatch**: The maximum number of mismatching letters allowed (see ?isMatching for the details). This man page focuses on exact matching of a constant width dictionary so max.mismatch=0 in the examples below. See ?matchPDict-inexact for inexact matching.
- **fixed**: If FALSE then IUPAC extended letters are interpreted as ambiguities (see ?isMatching for the details). This man page focuses on exact matching of a constant width dictionary so fixed=TRUE in the examples below. See ?matchPDict-inexact for inexact matching.
- **verbose**: TRUE or FALSE.

Details

In this man page, we assume that you know how to preprocess a dictionary of DNA patterns that can then be used with matchPDict, countPDict or whichPDict. Please see ?PDict if you don’t.

When using matchPDict, countPDict or whichPDict for exact matching of a constant width dictionary, the standard way to preprocess the original dictionary is by calling the PDict constructor on it with no extra arguments. This returns the preprocessed dictionary in a PDict object that can be used with matchPDict/countPDict/whichPDict.

Value

- matchPDict returns an MIndex object with length equal to the number of patterns in the pdict argument.
- countPDict returns an integer vector with length equal to the number of patterns in the pdict argument.
- whichPDict returns an integer vector made of the indices of the patterns in the pdict argument that have at least one match.

Author(s)

H. Pages

References


See Also

PDict-class, MIndex-class, matchPDict-inexact, isMatching, coverage, MIndex-method, matchPattern, alphabetFrequency, XStringViews-class, DNAString-class
Examples

```r
## A. A SIMPLE EXAMPLE OF EXACT MATCHING

## Creating the pattern dictionary:
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe$sequence)
dict0 # The original dictionary.
length(dict0) # Hundreds of thousands of patterns.
pdict0 <- PDict(dict0) # Store the original dictionary in
# a PDict object (preprocessing).

## Using the pattern dictionary on chromosome 3R:
library(BSgenome.Dmelanogaster.UCSC.dm3)
chr3R <- Dmelanogaster$chr3R # Load chromosome 3R
chr3R
mi0 <- matchPDict(pdict0, chr3R) # Search...

## Looking at the matches:
start_index <- startIndex(mi0) # Get the start index.
length(start_index) # Same as the original dictionary.
start_index[[8220]] # Starts of the 8220th pattern.
end_index <- endIndex(mi0) # Get the end index.
end_index[[8220]] # Ends of the 8220th pattern.
count_index <- countIndex(mi0) # Get the number of matches per pattern.
count_index[[8220]]
mi0[[8220]] # Get the matches for the 8220th pattern.
start(mi0[[8220]]) # Equivalent to startIndex(mi0)[[8220]].
sum(count_index) # Total number of matches.
table(count_index)
i0 <- which(count_index == max(count_index))
pdict0[[i0]] # The pattern with most occurrences.
mi0[[i0]] # Its matches as an IRanges object.
Views(chr3R, start=start_index[[i0]], end=end_index[[i0]]) # And as an XStringViews object.

## Get the coverage of the original subject:
 cov3R <- as.integer(coverage(mi0, 1, length(chr3R)))
max(cov3R)
mean(cov3R)
sum(cov3R != 0) / length(cov3R) # Only 2.44
if (interactive()) {
  plotCoverage <- function(coverage, start, end) {
    plot.new()
    plot.window(c(start, end), c(0, 20))
    axis(1)
    axis(2)
    axis(4)
    lines(start:end, coverage[start:end], type="l")
  }
  plotCoverage(cov3R, 27600000, 27900000)
}

## B. NAMING THE PATTERNS
```
### The names of the original patterns, if any, are propagated to the PDict and MIndex objects:

```r
dict0 <- mkAllStrings(letters, 4)[seq_len(length(dict0))]
dict0["abcd"]
pdict0n <- PDict(dict0)
names(pdict0n)[1:30]
pdict0n["abcd"]
mi0n <- matchPDict(pdict0n, chr3R)
names(mi0n)[1:30]
mi0n["abcd"]
```

This is particularly useful when unlisting an MIndex object:

```r
unlist(mi0)[1:10]
unlist(mi0n)[1:10] # keep track of where the matches are coming from
```

### C. PERFORMANCE

If getting the number of matches is what matters only (without regarding their positions), then countPDict() will be faster, especially when there is a high number of matches:

```r
count_index0 <- countPDict(pdict0, chr3R)
identical(count_index0, count_index) # TRUE
```

```r
if (interactive()) {
  ## What's the impact of the dictionary width on performance?
  ## Below is some code that can be used to figure out (will take a long time to run). For different widths of the original dictionary, we look at:
  ## o pptime: preprocessing time (in sec.) i.e. time needed for building the PDict object from the truncated input sequences;
  ## o nnodes: nb of nodes in the resulting Aho-Corasick tree;
  ## o nupatt: nb of unique truncated input sequences;
  ## o matchtime: time (in sec.) needed to find all the matches;
  ## o totalcount: total number of matches.
  getPDictStats <- function(dict, subject)
  {
    ans_width <- width(dict[1])
    ans_pptime <- system.time(pdict <- PDict(dict))[["elapsed"]]
    pptb <- pdict@threeparts@pptb
    ans_nnodes <- length(pptb@nodes)
    Biostrings:::ACTree.ints_per_acnode(pptb)
    ans_nupatt <- sum(!duplicated(pdict))
    ans_matchtime <- system.time{
      mi0 <- matchPDict(pdict, subject)
    }[["elapsed"]]
    ans_totalcount <- sum(countIndex(mi0))
    list(
      width=ans_width,
      pptime=ans_pptime,
      nnodes=ans_nnodes,
      nupatt=ans_nupatt,
      matchtime=ans_matchtime,
      totalcount=ans_totalcount
    )
  }
  ```
matchPDict-inexact

Inexact matching with matchPDict()/countPDict()/whichPDict()

Description

The matchPDict, countPDict and whichPDict functions efficiently find the occurrences in a text (the subject) of all patterns stored in a preprocessed dictionary.

This man page shows how to use these functions for inexact matching or when the original dictionary has a variable width.

See ?matchPDict for how to use these functions for exact matching of a constant width dictionary i.e. a dictionary where all the patterns have the same length (same number of nucleotides).
Details

In this man page, we assume that you know how to preprocess a dictionary of DNA patterns that can then be used with `matchPDict`, `countPDict` or `whichPDict`. Please see `?PDict` if you don’t.

When using `matchPDict`, `countPDict` or `whichPDict` for inexact matching or when the original dictionary has a variable width, a Trusted Band must be defined during the preprocessing step. This is done thru the `tb.start`, `tb.end` and `tb.width` arguments of the `PDict` constructor (see `?PDict` for the details).

Then `matchPDict/countPDict/whichPDict` can be called with a null or non-null `max.mismatch` value and the search for exact or inexact matches happens in 2 steps: (1) find all the exact matches of all the elements in the Trusted Band; then (2) for each element in the Trusted Band that has at least one exact match, compare the head and the tail of this element with the flanking sequences of the matches found in (1).

Note that the number of exact matches found in (1) will decrease exponentially with the width of the Trusted Band. Here is a simple guideline in order to get reasonably good performance: if `TBW` is the width of the Trusted Band (`TBW <- tb.width(pdict)`) and `L` the number of letters in the subject (`L <- nchar(subject)`), then `L / (4^TBW)` should be kept as small as possible, typically < 10 or 20.

In addition, when a Trusted Band has been defined during preprocessing, then `matchPDict/countPDict/whichPDict` can be called with `fixed=FALSE`. In this case, IUPAC extended letters in the head or the tail of the `PDict` object are treated as ambiguities.

Author(s)

H. Pages

References


See Also

`PDict-class`, `MIndex-class`, `matchPDict`

Examples

```r
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe$sequence)
dict0 # the original dictionary

## Preprocess the original dictionary by defining a Trusted Band that
## spans nucleotides 1 to 9 of each pattern.
pdict9 <- PDict(dict0, tb.end=9)
pdict9
tail(pdict9)
sum(duplicated(pdict9))
table(patternFrequency(pdict9))
```
matchPDict-inexact

```r
library(BSgenome.Dmelanogaster.UCSC.dm3)
chr3R <- Dmelanogaster$chr3R
table(countPDict(pdict9, chr3R, max.mismatch=1))
table(countPDict(pdict9, chr3R, max.mismatch=3))
table(countPDict(pdict9, chr3R, max.mismatch=5))
## ---------------------------------------------------------------------
## B. COMPARISON WITH EXACT MATCHING
## ---------------------------------------------------------------------
## When the original dictionary is of constant width, exact matching
## (i.e. 'max.mismatch=0' and 'fixed=TRUE) will be more efficient with
## a full-width Trusted Band (i.e. a Trusted Band that covers the entire
## dictionary) than with a Trusted Band of width < width(dict0).
pdict0 <- PDict(dict0)
count0 <- countPDict(pdict0, chr3R)
count0b <- countPDict(pdict9, chr3R, max.mismatch=0)
identical(count0b, count0) # TRUE
## ---------------------------------------------------------------------
## C. USING AN EXPLICIT TRUSTED BAND TO HANDLE A VARIABLE WIDTH
## DICTIONARY
## ---------------------------------------------------------------------
## Here is a small variable width dictionary that contains IUPAC
## ambiguities (pattern 1 and 3 contain an N):
dict0 <- DNAStringSet(c("TACCNG", "TAGT", "CGGNT", "AGTAG", "TAGT"))
## (Note that pattern 2 and 5 are identical.)
## If we only want to do exact matching, then it is recommended to use
## the widest possible Trusted Band i.e. to set its width to
## 'min(width(dict0))' because this is what will give the best
## performance. However, when 'dict0' contains IUPAC ambiguities (like
## in our case), it could be that one of them is falling into the
## Trusted Band so we get an error (only base letters can go in the
## Trusted Band for now):
## Not run:
PDict(dict0, tb.end=min(width(dict0))) # Error!
## End(Not run)
## In our case, the Trusted Band cannot be wider than 3:
pdict <- PDict(dict0, tb.end=3)
tail(pdict)
subject <- DNAString("TAGTACCAGGTTTGGG")
m <- matchPDict(pdict, subject)
countIndex(m) # pattern 2 and 5 have 1 exact match
m[[2]]
## We can take advantage of the fact that our Trusted Band doesn't cover
## the entire dictionary to allow inexact matching on the uncovered parts
## (the tail in our case):
m <- matchPDict(pdict, subject, fixed=FALSE)
```

---

```
```
A simple PWM matching function and related utilities

Description
A function implementing a simple algorithm for matching a set of patterns represented by a Position Weight Matrix (PWM) to a DNA sequence. PWM for amino acid sequences are not supported.

Usage

```r
matchPWM(pwm, subject, min.score="80%")
countPWM(pwm, subject, min.score="80%")
```

## Utility functions for basic manipulation of the Position Weight Matrix

```r
maxWeights(pwm)
maxScore(pwm)
reverseComplement(x, ...) # S4 method for matrix objects
```

Arguments

- **pwm**: A Position Weight Matrix (integer matrix with row names A, C, G and T).
- **subject**: A DNAString object containing the subject sequence.
- **min.score**: The minimum score for counting a match. Can be given as a percentage (e.g. "85%") of the highest possible score or as an integer.

Value

- **An XStringViews object** for matchPWM.
- A single integer for countPWM.
- An integer vector containing the max weight for each position in pwm for maxWeights.
- The highest possible score for a given Position Weight Matrix for maxScore.
- A PWM obtained by reverting the column order in PWM x and by reassigning each row to its complementary nucleotide for reverseComplement.
See Also

matchPattern, reverseComplement, DNAString-class, XStringViews-class

Examples

pwm <- rbind(A=c( 1, 0, 19, 20, 18, 1, 20, 7),
C=c(1, 0, 1, 0, 18, 0, 2),
G=c(17, 0, 0, 0, 1, 0, 3),
T=c(1, 20, 0, 0, 1, 0, 8))
maxWeights(pwm)
maxScore(pwm)
reverseComplement(pwm)

subject <- DNAString("AGTAAACAA")
PWMscore(pwm, subject, c(2:1, NA))

library(BSgenome.Dmelanogaster.UCSC.dm3)
chr3R <- unmasked(Dmelanogaster$chr3R)

## Match the plus strand
matchPWM(pwm, chr3R)
countPWM(pwm, chr3R)

## Match the minus strand
matchPWM(reverseComplement(pwm), chr3R)

matchPattern  String searching functions

Description

A set of functions for finding all the occurrences (aka "matches" or "hits") of a given pattern (typically short) in a (typically long) reference sequence or set of sequences (aka the subject)

Usage

matchPattern(pattern, subject, algorithm="auto",
max.mismatch=0, with.indels=FALSE, fixed=TRUE)
countPattern(pattern, subject, algorithm="auto",
max.mismatch=0, with.indels=FALSE, fixed=TRUE)
vmatchPattern(pattern, subject, algorithm="auto",
max.mismatch=0, with.indels=FALSE, fixed=TRUE)
vcountPattern(pattern, subject, algorithm="auto",
max.mismatch=0, with.indels=FALSE, fixed=TRUE)

Arguments

pattern  The pattern string.
subject  An XString, XStringViews or MaskedXString object for matchPattern and countPattern.
          An XStringSet or XStringViews object for vmatchPattern and vcountPattern.
**algorithm**  One of the following: "auto", "naive-exact", "naive-inexact", "boyer-moore", "shift-or" or "indels".

**max.mismatch**  The maximum number of mismatching letters allowed (see `isMatchingAt` for the details). If non-zero, an inexact matching algorithm is used.

**with.indels**  If TRUE then indels are allowed. In that case `max.mismatch` is interpreted as the maximum "edit distance" allowed between the pattern and a match. Note that in order to avoid pollution by redundant matches, only the "best local matches" are returned. Roughly speaking, a "best local match" is a match that is locally both the closest (to the pattern P) and the shortest. More precisely, a substring S' of the subject S is a "best local match" iff:

(a) \( \text{nedit}(P, S') \leq \text{max.mismatch} \)
(b) for every substring S1 of S':
   \( \text{nedit}(P, S1) > \text{nedit}(P, S') \)
(c) for every substring S2 of S that contains S':
   \( \text{nedit}(P, S2) \leq \text{nedit}(P, S') \)

One nice property of "best local matches" is that their first and last letters are guaranteed to be aligned with letters in P (i.e. they match letters in P).

**fixed**  If FALSE then IUPAC extended letters are interpreted as ambiguities (see `isMatchingAt` for the details).

**Details**

Available algorithms are: "naive exact", "naive inexact", "Boyer-Moore-like", "shift-or" and "indels". Not all of them can be used in all situations: restrictions depend on the length of the pattern, the class of the subject, and the values of `max.mismatch`, `with.indels` and `fixed`. All those parameters form the search criteria.

Note that the choice of an algorithm is not part of the search criteria. This is because algorithms are interchangeable, that is, if 2 different algorithms are compatible with a given search criteria, then choosing one over the other will not affect the result (but will most likely affect the performance). So there is no "wrong choice" of algorithm (strictly speaking).

Using `algorithm="auto"` is recommended because then the fastest algorithm will automatically be picked up among the set of compatible algorithms (if there is more than one).

**Value**

- An XStringViews object for `matchPattern`.
- A single integer for `countPattern`.
- An MIndex object for `vmatchPattern`.
- An integer vector for `vcountPattern`, with each element in the vector corresponding to the number of matches in the corresponding element of `subject`.

**Note**

Use `matchPDict` if you need to match a (big) set of patterns against a reference sequence.

Use `pairwiseAlignment` if you need to solve a (Needleman-Wunsch) global alignment, a (Smith-Waterman) local alignment, or an (ends-free) overlap alignment problem.
See Also

matchPDict, pairwiseAlignment, isMatchingAt, mismatch, matchLRPatterns, matchProbePair, maskMotif, alphabetFrequency, XStringViews-class, MIndex-class

Examples

```r
## ---------------------------------------------------------------------
## A. matchPattern()/countPattern()
## ---------------------------------------------------------------------

## A simple inexact matching example with a short subject:
x <- DNAString("AAGCGCGATATG")
m1 <- matchPattern("GCNNNAT", x)
m1
m2 <- matchPattern("GCNNNAT", x, fixed=FALSE)
as.matrix(m2)

## With DNA sequence of yeast chromosome number 1:
data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)
PpiI <- "GAACNNNNNCTC" # a restriction enzyme pattern
match1.PpiI <- matchPattern(PpiI, yeast1, fixed=FALSE)
match2.PpiI <- matchPattern(PpiI, yeast1, max.mismatch=1, fixed=FALSE)

## With a genome containing isolated Ns:
library(BSgenome.Celegans.UCSC.ce2)
chrII <- Celegans["chrII"]
alphabetFrequency(chrII)
m1 <- matchPattern("N", chrII)
m2 <- matchPattern("TGGGTGTCTTT", chrII) # no match
m2 <- matchPattern("TGGGTGTCTTT", chrII, fixed=FALSE) # 1 match

## Using wildcards ("N") in the pattern on a genome containing N-blocks:
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- maskMotif(Dmelanogaster$chrX, "N")
as(chrX, "XStringViews") # 4 non masked regions
matchPattern("TTATGNTTGTGA", chrX, fixed=FALSE)

## Can also be achieved with no mask:
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- maskMotif(Dmelanogaster$chrX, "N")
as(chrX, "XStringViews") # 4 non masked regions
matchPattern("TTATGNTTGTGA", chrX, fixed=FALSE) # 1 match

## B. vmatchPattern()/vcountPattern()
## ---------------------------------------------------------------------

Ebox <- DNAString("CANNTG")
subject <- Celegans$upstream5000
mindex <- vmatchPattern(Ebox, subject, fixed=FALSE)
count_index <- countIndex(mindex) # Get the number of matches per
    # subject element.
simultaneous = TRUE # Total number of matches.
table(count_index) # The subject element with most matches.
```
matchProbePair

Find "theoretical amplicons" mapped to a probe pair

Description

In the context of a computer-simulated PCR experiment, one wants to find the amplicons mapped to a given primer pair. The matchProbePair function can be used for this: given a forward and a reverse probe (i.e. the chromosome-specific sequences of the forward and reverse primers used for the experiment) and a target sequence (generally a chromosome sequence), the matchProbePair function will return all the "theoretical amplicons" mapped to this probe pair.

Usage

matchProbePair(Fprobe, Rprobe, subject, algorithm="auto", logfile=NULL, verbose=FALSE)
Arguments

Fprobe  The forward probe.
Rprobe  The reverse probe.
subject A DNAString object (or an XStringViews object with a DNAString subject) containing the target sequence.
algorithm One of the following: "auto", "naive-exact", "naive-inexact", "boyer-moore" or "shift-or". See matchPattern for more information.
logfile A file used for logging.
verbose TRUE or FALSE.

Details

The matchProbePair function does the following: (1) find all the "plus hits" i.e. the Fprobe and Rprobe matches on the "plus" strand, (2) find all the "minus hits" i.e. the Fprobe and Rprobe matches on the "minus" strand and (3) from the set of all (plus_hit, minus_hit) pairs, extract and return the subset of "reduced matches" i.e. the (plus_hit, minus_hit) pairs such that (a) plus_hit <= minus_hit and (b) there are no hits (plus or minus) between plus_hit and minus_hit. This set of "reduced matches" is the set of "theoretical amplicons".

Value

An XStringViews object containing the set of "theoretical amplicons".

Author(s)

H. Pages

See Also

matchPattern, matchLRPatterns, findPalindromes, reverseComplement, XStringViews

Examples

library(BSgenome.Dmelanogaster.UCSC.dm3)
subject <- Dmelanogaster$chr3R

## With 20-nucleotide forward and reverse probes:
Fprobe <- "AGCTCCGAGTTCTGCAATA"
Rprobe <- "GTTGTTCACAAATATGCGG"
matchProbePair(Fprobe, Rprobe, subject) # 1 "theoretical amplicon"

## With shorter forward and reverse probes, the risk of having multiple ## "theoretical amplicons" increases:
Fprobe <- "AGCTCCGAGTCC"
Rprobe <- "GTTGTTCACAA"
matchProbePair(Fprobe, Rprobe, subject) # 2 "theoretical amplicons"
Fprobe <- "AGCTCCAGTT"
Rprobe <- "GTTGTTCACA"
matchProbePair(Fprobe, Rprobe, subject) # 9 "theoretical amplicons"
Description

Simple gap implementation of Needleman-Wunsch global alignment algorithm.

Usage

needwunsQS(s1, s2, substmat, gappen = 8)

Arguments

s1, s2  an R character vector of length 1 or an XString object.
substmat  matrix of alignment score values.
gappen  penalty for introducing a gap in the alignment.

Details

Follows specification of Durbin, Eddy, Krogh, Mitchison (1998). This function has been deprecated and is being replaced by pairwiseAlignment.

Value

An instance of class "PairwiseAlignedFixedSubject".

Author(s)

Vince Carey (stvjc@channing.harvard.edu) (original author) and H. Pages (current maintainer).

References


See Also

pairwiseAlignment, PairwiseAlignedFixedSubject-class, substitution.matrices

Examples

## Not run:
## This function has been deprecated
## Use 'pairwiseAlignment' instead.

## nucleotide alignment
mat <- matrix(-5L, nrow = 4, ncol = 4)
for (i in seq_len(4)) mat[i, i] <- 0L
rownames(mat) <- colnames(mat) <- DNA_ALPHABET[1:4]
s1 <- DNAString(paste(sample(DNA_ALPHABET[1:4], 1000, replace=TRUE), collapse="
")
s2 <- DNAString(paste(sample(DNA_ALPHABET[1:4], 1000, replace=TRUE), collapse="
")
nw0 <- needwunsQS(s1, s2, mat, gappen = 0)
nw1 <- needwunsQS(s1, s2, mat, gappen = 1)
nw5 <- needwunsQS(s1, s2, mat, gappen = 5)

## amino acid alignment
needwunsQS("PAWHEAE", "HEAGAWGHEE", substmat = "BLOSUM50")

## End(Not run)

pairwiseAlignment

### Optimal Pairwise Alignment

#### Description

Solves (Needleman-Wunsch) global alignment, (Smith-Waterman) local alignment, and (ends-free) overlap alignment problems.

#### Usage

```r
pairwiseAlignment(pattern, subject, ...)
```  
S4 method for signature 'XStringSet, XStringSet':

```r
pairwiseAlignment(pattern, subject,
  patternQuality = PhredQuality(22L), subjectQuality = PhredQuality(22L),
  type = "global", substitutionMatrix = NULL, fuzzyMatrix = NULL,
  gapOpening = -10, gapExtension = -4, scoreOnly = FALSE)
```  
S4 method for signature 'QualityScaledXStringSet, QualityScaledXStringSet':

```r
pairwiseAlignment(pattern, subject,
  type = "global", substitutionMatrix = NULL, fuzzyMatrix = NULL,
  gapOpening = -10, gapExtension = -4, scoreOnly = FALSE)
```  
#### Arguments

- `pattern`: a character vector of any length, an `XString`, or an `XStringSet` object.
- `subject`: a character vector of length 1 or an `XString` object.
- `patternQuality`: objects of class `XStringQuality` representing the respective quality scores for `pattern` and `subject` that are used in a quality-based method for generating a substitution matrix. These two arguments are ignored if `!is.null(substitutionMatrix)` or if its respective string set (`pattern`, `subject`) is of class `QualityScaledXStringSet`.
- `subjectQuality`: objects of class `XStringQuality` representing the respective quality scores for `pattern` and `subject` that are used in a quality-based method for generating a substitution matrix. These two arguments are ignored if `!is.null(substitutionMatrix)` or if its respective string set (`pattern`, `subject`) is of class `QualityScaledXStringSet`.
- `type`: type of alignment. One of "global", "local", "overlap", "patternOverlap", and "subjectOverlap" where "global" = align whole strings with end gap penalties, "local" = align string fragments, "overlap" = align whole strings without end gap penalties, "patternOverlap" = align whole strings without end gap penalties on `pattern` and with end gap penalties on `subject`, "subjectOverlap" = align whole strings with end gap penalties on `pattern` and without end gap penalties on `subject`.
- `substitutionMatrix`: substitution matrix for a non-quality based alignment. It cannot be used in conjunction with `patternQuality` and `subjectQuality` arguments.
- `fuzzyMatrix`: fuzzy match matrix for quality-based alignments. It takes values between 0 and 1; where 0 is an unambiguous mismatch, 1 is an unambiguous match, and values in between represent a fraction of "matchiness".
pairwiseAlignment

- **gapOpening**: the cost for opening a gap in the alignment.
- **gapExtension**: the incremental cost incurred along the length of the gap in the alignment.
- **scoreOnly**: logical to denote whether or not to return just the scores of the optimal pairwise alignment.
- **...**: optional arguments to generic function to support additional methods.

**Details**

If `scoreOnly == FALSE`, the pairwise alignment with the maximum alignment score is returned. If more than one pairwise alignment has the maximum alignment score exists, the first alignment along the subject is returned. If there are multiple pairwise alignments with the maximum alignment score at the chosen subject location, then at each location along the alignment mismatches are given preference to insertions/deletions. For example, pattern: [1] ATTA; subject: [1] AT-A is chosen above pattern: [1] ATTA; subject: [1] A-TA if they both have the maximum alignment score.


**Value**

If `scoreOnly == FALSE`, an instance of class `PairwiseAlignedFixedSubject` is returned. If `scoreOnly == TRUE`, a numeric vector containing the scores for the optimal pairwise alignments is returned.

**Note**

Use `matchPattern` or `vmatchPattern` if you need to find all the occurrences (eventually with indels) of a given pattern in a reference sequence or set of sequences.

Use `matchPDict` if you need to match a (big) set of patterns against a reference sequence.

**Author(s)**

P. Aboyoun and H. Pages

**References**


**See Also**

`stringDist`, `PairwiseAlignedFixedSubject-class`, `XStringQuality-class`, `substitution.matrices`, `matchPattern`
Examples

```r
## Nucleotide global, local, and overlap alignments
s1 <- DNAString("ACTTCACCAGCTCCCTGGCGGTAAGTTGATCAAAGGAAACGCAAAGTTTTCAAG")
s2 <- DNAString("GTTTCACTACTTCTTCCGGTAAAGTAAATATATAAATATATAAAAAATATATAATTTCATC")
# First use a fixed substitution matrix
mat <- nucleotideSubstitutionMatrix(match = 1, mismatch = -3, baseOnly = TRUE)
globalAlign <- pairwiseAlignment(s1, s2, substitutionMatrix = mat, gapOpening = -5, gapExtension = -2)
localAlign <- pairwiseAlignment(s1, s2, type = "local", substitutionMatrix = mat, gapOpening = -5, gapExtension = -2)
overlapAlign <- pairwiseAlignment(s1, s2, type = "overlap", substitutionMatrix = mat, gapOpening = -5, gapExtension = -2)
# Then use quality-based method for generating a substitution matrix
pairwiseAlignment(s1, s2,
                patternQuality = SolexaQuality(rep(c(22L, 12L), times = c(36, 18))),
                subjectQuality = SolexaQuality(rep(c(22L, 12L), times = c(40, 20))),
                scoreOnly = TRUE)
## Amino acid global alignment
pairwiseAlignment(AAString("PAWHEAE"), AAString("HEAGAWGHEE"), substitutionMatrix = "BLOSUM50",
                gapOpening = 0, gapExtension = -8)
```

---

### phiX174Phage

**Versions of bacteriophage phiX174 complete genome and sample short reads**

---

### Description

Six versions of the complete genome for bacteriophage φX174 as well as a small number of Solexa short reads, qualities associated with those short reads, and counts for the number times those short reads occurred.

### Details

The `phiX174Phage` object is a `DNAStringSet` containing the following six naturally occurring versions of the bacteriophage φX174 genome cited in Smith et al.:

- **Genbank**: The version of the genome from GenBank (NC_001422.1, GI:9626372).
- **RF70s**: A preparation of φ X double-stranded replicative form (RF) of DNA by Clyde A. Hutchison III from the late 1970s.
- **SS78**: A preparation of φ X virion single-stranded DNA from 1978.
- **Bull**: The sequence of wild-type φ X used by Bull et al.
- **G’97**: The φ X replicative form (RF) of DNA from Bull et al.
- **NEB’03**: A φ X replicative form (RF) of DNA from New England BioLabs (NEB).

The `srPhiX174` object is a `DNAStringSet` containing short reads from a Solexa machine.

The `quPhiX174` object is a `BStringSet` containing Solexa quality scores associated with `srPhiX174`.

The `wtPhiX174` object is an integer vector containing counts associated with `srPhiX174`.
References

http://www.genome.jp/dbget-bin/www_bget?refseq+NC_001422


Examples

data(phiX174Phage)
nchar(phiX174Phage)
genBankPhage <- phiX174Phage[[1]]
genBankSubstring <- substring(genBankPhage, 2793-34, 2811+34)

data(srPhiX174)
srPhiX174
quPhiX174
summary(wtPhiX174)

alignPhiX174 <-
  pairwiseAlignment(srPhiX174, genBankSubstring,
                  patternQuality = SolexaQuality(quPhiX174),
                  subjectQuality = SolexaQuality(99L),
                  type = "subjectOverlap")

summary(alignPhiX174, weight = wtPhiX174)

---

pid

Percent Sequence Identity

Description

Calculates the percent sequence identity for a pairwise sequence alignment.

Usage

pid(x, type="PID1")

Arguments

x    
a \text{PairwiseAlignedFixedSubject} object.

\text{type}    
one of percent sequence identity. One of "PID1", "PID2", "PID3", and "PID4". See Details for more information.

Details

Since there is no universal definition of percent sequence identity, the \text{pid} function calculates this statistic in the following types:

"\text{PID1}": 100 * (identical positions) / (aligned positions + internal gap positions)
"PID2": $100 \times \text{(identical positions)} / (\text{aligned positions})$

"PID3": $100 \times \text{(identical positions)} / (\text{length shorter sequence})$

"PID4": $100 \times \text{(identical positions)} / (\text{average length of the two sequences})$

Value
A numeric vector containing the specified sequence identity measures.

Author(s)
P. Aboyoun

References

See Also
pairwiseAlignment, PairwiseAlignedFixedSubject-class, match-utils

Examples
```r
s1 <- DNAString("AGTATAGATGATAGAT")
s2 <- DNAString("AGTAGATGGATGATAGATA")
palign1 <- pairwiseAlignment(s1, s2)
palign1
pid(palign1)

palign2 <- pairwiseAlignment(s1, s2,
   substitutionMatrix =
   nucleotideSubstitutionMatrix(match = 2, mismatch = 10, baseOnly = TRUE))
palign2
pid(palign2, type = "PID4")
```

pmatchPattern Longest Common Prefix/Suffix/Substring searching functions

Description
Functions for searching the Longest Common Prefix/Suffix/Substring of two strings.
WARNING: These functions are experimental and might not work properly! Full documentation will come later.
Please send questions/comments to hpages@fhcrc.org
Thanks for your comprehension!
Usage

```
lcprefix(s1, s2)
lcsuffix(s1, s2)
lcsubstr(s1, s2)
```

```
pmatchPattern(pattern, subject, maxlength.out=1L)
```

Arguments

- **s1**: 1st string, a character string or an XString object.
- **s2**: 2nd string, a character string or an XString object.
- **pattern**: The pattern string.
- **subject**: An XString object containing the subject string.
- **maxlength.out**: The maximum length of the output i.e. the maximum number of views in the returned object.

See Also

```
matchPattern, XStringViews-class, XString-class
```

---

**readFASTA**

*Functions to read/write FASTA formatted files*

**Description**

FASTA is a simple file format for biological sequence data. A file may contain one or more sequences, for each sequence there is a description line which begins with a >.

**Usage**

```
fasta.info(file, use.descs=TRUE)
readFASTA(file, checkComments=TRUE, strip.descs=TRUE)
writeFASTA(x, file="", width=80)
```

**Arguments**

- **file**: Either a character string naming a file or a connection open for reading or writing. If "" (the default for writeFASTA), then the function writes to the standard output connection (the console) unless redirected by sink.
- **use.descs**: TRUE or FALSE. Whether or not the description lines should be used to name the elements of the returned integer vector.
- **checkComments**: Whether or not comments, lines beginning with a semi-colon should be found and removed.
- **strip.descs**: Whether or not the ">" marking the beginning of the description lines should be removed. Note that this argument is new in Biostrings >= 2.8. In previous versions readFASTA was keeping the ">".
- **x**: A list as one returned by readFASTA.
- **width**: The maximum number of letters per line of sequence.
Details

FASTA is a widely used format in biology. It is a relatively simple markup. I am not aware of a standard. It might be nice to check to see if the data that were parsed are sequences of some appropriate type, but without a standard that does not seem possible.

There are many other packages that provide similar, but different capabilities. The one in the package seqinr seems most similar but they separate the biological sequence into single character strings, which is too inefficient for large problems.

Value

An integer vector (for `fasta.info`) or a list (for `readFASTA`) with one element for each sequence in the file. For `readFASTA`, the elements are in two parts, one the description and the second a character string of the biological sequence.

Author(s)

R. Gentleman, H. Pages

See Also

`read.BStringSet`, `read.DNASTingSet`, `read.RNAStringSet`, `read.AAStringSet`, `write.XStringSet`, `read.table`, `scan`, `write.table`

Examples

```r
f1 <- system.file("extdata", "someORF.fa", package="Biostrings")
file.info(f1)
ff <- readFASTA(f1, strip.descs=TRUE)
desc <- sapply(ff, function(x) x$desc)
## Keep the "reverse complement" sequences only
ff2 <- ff[grep("reverse complement", desc, fixed=TRUE)]
writeFASTA(ff2, file.path(tempdir(), "someORF2.fa"))
```

---

replaceLetterAt

**Replacing letters in a sequence at some specified locations**

Description

`replaceLetterAt` first makes a copy of a sequence and then replaces the original letters by new letters at some specified locations in the copied sequence.

`.inplaceReplaceLetterAt` is the IN PLACE version of `replaceLetterAt`: it will modify the original sequence in place i.e. without copying it first. Note that in place modification of a sequence is fundamentally dangerous because it alters all objects defined in your session that make reference to the modified sequence. NEVER use `.inplaceReplaceLetterAt`, unless you know what you are doing!

Usage

```r
replaceLetterAt(x, at, letter, if.not.extending="replace", verbose=FALSE)
```

## NEVER USE THIS FUNCTION!

`.inplaceReplaceLetterAt(x, at, letter)`
replaceLetterAt

Arguments

x  A DNAString object.

at  An integer vector with no NAs specifying the locations where the replacements must occur. Note that locations can be repeated and in this case the last replacement to occur at a given location prevails.

letter  Character vector with no NAs. The total number of letters in letter ($\sum(nchar(letter))$) must be equal to the number of locations ($\text{length(at)}$).

if.not.extending  What to do if the new letter is not "extending" the old letter? The new letter "extends" the old letter if both are IUPAC letters and the new letter is as specific or less specific than the old one (e.g. M extends A, Y extends Y, but Y doesn't extend S). Possible values are "replace" (the default) for replacing in all cases, "skip" for not replacing when the new letter does not extend the old letter, "merge" for merging the new IUPAC letter with the old one, and "error" for raising an error.

Note that the gap ("-")) and hard masking ("+") letters are not extending or extended by any other letter.

Also note that "merge" is the only value for the if.not.extending argument that guarantees the final result to be independent on the order the replacement is performed (although this is only relevant when at contains duplicated locations, otherwise the result is of course always independent on the order, whatever the value of if.not.extending is).

verbose  When TRUE, a warning will report the number of skipped or merged letters.

Details

.inplaceReplaceLetterAt semantic is equivalent to calling replaceLetterAt with if.not.extending="merge" and verbose=FALSE.

Never use .inplaceReplaceLetterAt! It is used by the injectSNPs function in the BSgenome package, as part of the "lazy sequence loading" mechanism, for altering the original sequences of a BSgenome object at "sequence-load time". This alteration consists in injecting the IUPAC ambiguity letters representing the SNPs into the just loaded sequence, which is the only time where in place modification of the external data of an XString object is safe.

Value

A DNAString object of the same length as the original object x for replaceLetterAt.

Author(s)

H. Pages

See Also

IUPAC_CODE_MAP, chartr, injectHardMask, DNAString, injectSNPs, BSgenome

Examples

replaceLetterAt(DNAString("AAMAA"), c(5, 1, 3, 1), "TYNC")
replaceLetterAt(DNAString("AAMAA"), c(5, 1, 3, 1), "TYNC", if.not.extending="merge")
reverseComplement  Sequence reversing and complementing

Description

Use these functions for reversing a sequence and/or complementing a DNA sequence.

Usage

```r
## S4 method for signature 'XString':
reverse(x, ...)
complement(x, ...)
reverseComplement(x, ...)
```

Arguments

- `x` An `IRanges`, `NormalIRanges`, `MaskCollection`, `XString`, `XStringSet`, `XStringViews` or `MaskedXString` object for `reverse`.
  A `DNAString`, `RNAString`, `DNAStringSet`, `RNAStringSet`, `XStringViews` (with `DNAString` or `RNAString` subject), `MaskedDNAString` or `MaskedRNAString` object for `complement` and `reverseComplement`.
  Additional arguments to be passed to or from methods.

Details

Given an `XString` object `x`, `reverse(x)` returns an object of the same `XString` subtype as `x` where letters in `x` have been reordered in the reverse order.

If `x` is a `DNAString` or `RNAString` object, `complement(x)` returns an object where each base in `x` is "complemented" i.e. A, C, G, T in a `DNAString` object are replaced by T, G, C, A respectively and A, C, G, U in a `RNAString` object are replaced by U, G, C, A respectively.

Letters belonging to the 'IUPAC extended genetic alphabet' are also replaced by their complement (M <-> K, R <-> Y, S <-> S, V <-> B, W <-> W, H <-> D, N <-> N) and the gap ("-") and hard masking (+) letters are unchanged.

`reverseComplement(x)` is equivalent to `reverse(complement(x))` but is faster and more memory efficient.

Value

An object of the same class and length as the original object.

See Also

- `IRanges-class`, `NormalIRanges-class`, `MaskCollection-class`, `DNAString-class`, `RNAString-class`,
- `DNAStringSet-class`, `RNAStringSet-class`, `XStringViews-class`, `MaskedXString-class`, `strrev`, `chartr`,
- `findPalindromes`
Examples

```r
## A. SIMPLE EXAMPLES
x <- DNAString("ACGT-YN-")
reverseComplement(x)

library(drosophila2probe)
x <- DNAStringSet(drosophila2probe$sequence)
x
alphabetFrequency(x, collapse=TRUE)
x
rcx <- reverseComplement(x)
rcx
alphabetFrequency(rcx, collapse=TRUE)

## B. SEARCHING THE REVERSE STRAND OF A CHROMOSOME
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- Dmelanogaster$chrX
pattern <- DNAString("ACCAACNNGGTTG")
matchPattern(pattern, chrX, fixed=FALSE) # 3 hits on strand +
rcpattern <- reverseComplement(pattern)
rcpattern
m0 <- matchPattern(rcpattern, chrX, fixed=FALSE)
m0 # 5 hits on strand -

## Applying reverseComplement() to the subject instead of the pattern is not
## a good idea for 2 reasons:
## (1) Chromosome sequences are generally big and sometimes very big
## so computing the reverse complement of the positive strand will
## take time and memory proportional to its length.
chrXminus <- reverseComplement(chrX) # needs to allocate 22M of memory!

## (2) Chromosome locations are generally given relatively to the positive
## strand, even for features located in the negative strand, so after
## doing this:

m1 <- matchPattern(pattern, chrXminus, fixed=FALSE)

## the start/end of the matches are now relative to the negative strand.
## You need to apply reverseComplement() again on the result if you want
## them to be relative to the positive strand:

m2 <- reverseComplement(m1) # allocates 22M of memory, again!

## and finally to apply rev() to sort the matches from left to right
## (5'3' direction) like in m0:

m3 <- rev(m2) # same as m0, finally!

## WARNING: Before you try the example below on human chromosome 1, be aware
## that it will require the allocation of about 500Mb of memory!
if (interactive()) {
  library(BSgenome.Hsapiens.UCSC.hg18)
  chr1 <- Hsapiens$chr1
```
stringDist

String Distance/Alignment Score Matrix

Description

Computes the Levenshtein edit distance or pairwise alignment score matrix for a set of strings.

Usage

stringDist(x, method = "levenshtein", ignoreCase = FALSE, diag = FALSE, upper = FALSE)

## S4 method for signature 'XStringSet':
stringDist(x, method = "levenshtein", ignoreCase = FALSE, diag = FALSE, upper = FALSE, type = "global", quality = PhredQuality(22L), substitutionMatrix = NULL, fuzzyMatrix = NULL, gapOpening = 0, gapExtension = -1)

## S4 method for signature 'QualityScaledXStringSet':
stringDist(x, method = "quality", ignoreCase = FALSE, diag = FALSE, upper = FALSE, type = "global", substitutionMatrix = NULL, fuzzyMatrix = NULL, gapOpening = 0, gapExtension = -1)

Arguments

x : a character vector or an XStringSet object.

method : calculation method. One of "levenshtein", "quality", or "substitutionMatrix".

ignoreCase : logical value indicating whether to ignore case during scoring.

diag : logical value indicating whether the diagonal of the matrix should be printed by print.dist.

upper : logical value indicating whether the diagonal of the matrix should be printed by print.dist.

type : type of alignment. One of "global", "local", and "overlap", where "global" = align whole strings with end gap penalties, "local" = align string fragments, "overlap" = align whole strings without end gap penalties. This argument is ignored if method == "levenshtein".

quality : object of class XStringQuality representing the quality scores for x that are used in a quality-based method for generating a substitution matrix. This argument is ignored if method != "quality".

substitutionMatrix : symmetric substitution matrix for a non-quality based alignment. This argument is ignored if method != "substitutionMatrix".

fuzzyMatrix : fuzzy match matrix for quality-based alignments. It takes values between 0 and 1; where 0 is an unambiguous mismatch, 1 is an unambiguous match, and values in between represent a fraction of "matchiness".

gapOpening : penalty for opening a gap in the alignment. This argument is ignored if method == "levenshtein".

gapExtension : penalty for extending a gap in the alignment. This argument is ignored if method == "levenshtein".

... : optional arguments to generic function to support additional methods.
subXString

Details

Uses the underlying pairwiseAlignment code to compute the distance/alignment score matrix.

Value

Returns an object of class "dist".

Author(s)

P. Aboyoun

See Also

dist, agrep, pairwiseAlignment, substitution.matrices

Examples

stringDist(c("lazy", "HaZy", "crAzY"))
stringDist(c("lazy", "HaZy", "crAzY"), ignoreCase = TRUE)

data(phiX174Phage)
plot(hclust(stringDist(phiX174Phage), method = "single"))

data(srPhiX174)
stringDist(srPhiX174[1:4])
stringDist(srPhiX174[1:4], method = "quality",
  quality = SolexaQuality(quPhiX174[1:4]),
  gapOpening = -10, gapExtension = -4)

subXString  Fast substring extraction

Description

Functions for fast substring extraction.

Usage

subXString(x, start=NA, end=NA, length=NA)
substr(x, start=NA, stop=NA)
substring(text, first=NA, last=NA)

Arguments

x    An XString object for subXString. A character vector, an XStringViews,
     XString, or MaskedXString object for substr or substring.
start A numeric vector.
end   A numeric vector.
length A numeric vector.
stop  A numeric vector.
text  A character vector, an XStringViews or an XString object.
first A numeric vector.
last  A numeric vector.
Details

subXString is deprecated in favor of subseq.

Value

An XString object of the same subtype as x for subXString.
A character vector for substr and substring.

See Also

subseq, letter, XString-class, XStringViews-class

substitution.matrices

Scoring matrices

Description

Predefined substitution matrices for nucleotide and amino acid alignments.

Usage

data(BLOSUM45)
data(BLOSUM50)
data(BLOSUM62)
data(BLOSUM80)
data(BLOSUM100)
data(PAM30)
data(PAM40)
data(PAM70)
data(PAM120)
data(PAM250)
nucleotideSubstitutionMatrix(match = 1, mismatch = 0, baseOnly = FALSE, type = "DNA")
qualitySubstitutionMatrices(fuzzyMatch = c(0, 1), alphabetLength = 4L, qualityClass = "Phred.quality", bitScale = 1)
errorSubstitutionMatrices(errorProbability, fuzzyMatch = c(0, 1), alphabetLength = 4L, bitScale = 1)

Arguments

match the scoring for a nucleotide match.
mismatch the scoring for a nucleotide mismatch.
baseOnly TRUE or FALSE. If TRUE, only uses the letters in the "base" alphabet i.e. "A", "C", "G", "T".
type either "DNA" or "RNA".
fuzzyMatch a named or unnamed numeric vector representing the base match probability.
errorProbability a named or unnamed numeric vector representing the error probability.
alphabetLength an integer representing the number of letters in the underlying string alphabet. For DNA and RNA, this would be 4L. For Amino Acids, this could be 20L.
qualityClass a character string of either "PhredQuality" or "SolexaQuality".

bitScale a numeric value to scale the quality-based substitution matrices. By default, this is 1, representing bit-scale scoring.

Format

The BLOSUM and PAM matrices are square symmetric matrices with integer coefficients, whose row and column names are identical and unique: each name is a single letter representing a nucleotide or an amino acid.

nucleotideSubstitutionMatrix produces a substitution matrix for all IUPAC nucleic acid codes based upon match and mismatch parameters.

errorSubstitutionMatrices produces a two element list of numeric square symmetric matrices, one for matches and one for mismatches.

qualitySubstitutionMatrices produces the substitution matrices for Phred or Solexa quality-based reads.

Details

The BLOSUM and PAM matrices are not unique. For example, the definition of the widely used BLOSUM62 matrix varies depending on the source, and even a given source can provide different versions of "BLOSUM62" without keeping track of the changes over time. NCBI provides many matrices here ftp://ftp.ncbi.nih.gov/blast/matrices/ but their definitions don’t match those of the matrices bundled with their stand-alone BLAST software available here ftp://ftp.ncbi.nih.gov/blast/
The BLOSUM45, BLOSUM62, BLOSUM80, PAM30 and PAM70 matrices were taken from NCBI stand-alone BLAST software.
The BLOSUM50, BLOSUM100, PAM40, PAM120 and PAM250 matrices were taken from ftp://ftp.ncbi.nih.gov/blast/matrices/
The quality matrices computed in qualitySubstitutionMatrices are based on the paper by Ketil Malde. Let $\epsilon_i$ be the probability of an error in the base read. For "Phred" quality measures $Q$ in $[0, 99]$, these error probabilities are given by $\epsilon_i = 10^{-Q/10}$. For "Solexa" quality measures $Q$ in $[-5, 99]$, they are given by $\epsilon_i = 1 - 1/(1 + 10^{-Q/10})$. Assuming independence within and between base reads, the combined error probability of a mismatch when the underlying bases do match is $\epsilon_c = \epsilon_1 + \epsilon_2 - (n/(n-1)) \times \epsilon_1 \times \epsilon_2$, where $n$ is the number of letters in the underlying alphabet. Using $\epsilon_c$, the substitution score is given by when two bases match is given by $b \times \log_2(\gamma_{x,y} \times (1 - \epsilon_c) \times n + (1 - \gamma_{x,y}) \times \epsilon_c \times (n/(n-1)))$, where $b$ is the bit-scaling for the scoring and $\gamma_{x,y}$ is the probability that characters $x$ and $y$ represents the same underlying information (e.g. using IUPAC, $\gamma_{A,A} = 1$ and $\gamma_{A,N} = 1/4$. In the arguments listed above fuzzyMatch represents $\gamma_{x,y}$ and errorProbability represents $\epsilon_i$.

Author(s)

H. Pages and P. Aboyoun

References


See Also

groupAlignment, GroupAlignmentFixedSubject-class, DNAString-class, AAStrong-class, PhredQuality-class, SolexaQuality-class
Examples

```r
s1 <- DNAString("ACTTCACCAGCTCCCTGGCGGTAAGTTGATCAAAGGAAACGCAAAGTTTTCAAG")
s2 <- DNAString("GTTTCACTACTTCCTTTCGGGTAAGTAAATATATAAATATATAAAAATATATTTTCATC")

## Fit a global pairwise alignment using edit distance scoring
pairwiseAlignment(s1, s2,
substitutionMatrix = nucleotideSubstitutionMatrix(0, -1, TRUE),
gapOpening = 0, gapExtension = -1)

## Examine quality-based match and mismatch bit scores for DNA/RNA
## strings in pairwiseAlignment.
## By default patternQuality and subjectQuality are PhredQuality(22L).
qualityMatrices <- qualitySubstitutionMatrices()
qualityMatrices["22", "22", "i"]
qualityMatrices["22", "22", "o"]

pairwiseAlignment(s1, s2)

## Get the substitution scores when the error probability is 0.1
subscores <- errorSubstitutionMatrices(errorProbability = 0.1)
submat <- matrix(subscores[,"o"], 4, 4)
diag(submat) <- subscores[,"i"]
dimnames(submat) <- list(DNA_ALPHABET[1:4], DNA_ALPHABET[1:4])
submat

pairwiseAlignment(s1, s2, substitutionMatrix = submat)

## Align two amino acid sequences with the BLOSUM62 matrix
aa1 <- AAString("HXBLVYMGLCHFDCXVEHIKQZ")
aa2 <- AAString("QRNYMYCFQCISGNEYKQN")
pairwiseAlignment(aa1, aa2, substitutionMatrix = "BLOSUM62", gapOpening = -3, gapExtension = -1)

## See how the gap penalty influences the alignment
pairwiseAlignment(aa1, aa2, substitutionMatrix = "BLOSUM62", gapOpening = -6, gapExtension = -2)

## See how the substitution matrix influences the alignment
pairwiseAlignment(aa1, aa2, substitutionMatrix = "BLOSUM50", gapOpening = -3, gapExtension = -1)

data(BLOSUM62)
BLOSUM62["Q", "Z"]
b62 <- as.matrix(read.table(file, check.names=FALSE))
b62["Q", "Z"]
```

---

toComplex

### Turning a DNA sequence into a vector of complex numbers

**Description**

The `toComplex` utility function turns a `DNAString` object into a complex vector.
Usage
toComplex(x, baseValues)

Arguments
x
A DNAString object.
baseValues
A named complex vector containing the values associated to each base e.g. c(A=1+0i, G=0+1i, T=-1+0i, C=0-1i)

Value
A complex vector of the same length as x.

Author(s)
H. Pages

See Also
DNAString

Examples
seq <- DNAString("accacgtaccattgtcct")
baseValues1 <- c(A=1+0i, G=0+1i, T=-1+0i, C=0-1i)
toComplex(seq, baseValues1)

## GC content:
baseValues2 <- c(A=0, C=1, G=1, T=0)
sum(as.integer(toComplex(seq, baseValues2)))
## Note that there are better ways to do this (see ?alphabetFrequency)

translate DNA/RNA transcription and translation

Description
Functions for transcription and/or translation of DNA or RNA sequences, and related utilities.

Usage
transcribe(x)
cDNA(x)
codons(x)
translate(x)

## Related utilities
dna2rna(x)
rna2dna(x)
Arguments

x A DNAString object for transcribe and dna2rna.
An RNAString object for cDNA and rna2dna.
A DNAString, RNAString, MaskedDNAString or MaskedRNAString object for codons.
A DNAString, RNAString, DNAStringSet, RNAStringSet, MaskedDNAString or MaskedRNAString object for translate.

Details

transcribe reproduces the biological process of DNA transcription that occurs in the cell.
cDNA reproduces the process of synthesizing complementary DNA from a mature mRNA template.
translate reproduces the biological process of RNA translation that occurs in the cell. The input of the function can be either RNA or coding DNA. The Standard Genetic Code (see ?GENETIC_CODE) is used to translate codons into amino acids. codons is a utility for extracting the codons involved in this translation without translating them.
dna2rna and rna2dna are low-level utilities for converting sequences from DNA to RNA and vice-versa. All what this conversion does is to replace each occurrence of T by a U and vice-versa.

Value

An RNAString object for transcribe and dna2rna.
A DNAString object for cDNA and rna2dna.

Note that if the sequence passed to transcribe or cDNA is considered to be oriented 5’-3’, then the returned sequence is oriented 3’-5’.

An XStringViews object with 1 view per codon for codons. When x is a MaskedDNAString or MaskedRNAString object, its masked parts are interpreted as introns and filled with the + letter in the returned object. Therefore codons that span across masked regions are represented by views that have a width > 3 and contain the + letter. Note that each view is guaranteed to contain exactly 3 base letters.

An AAString object for translate.

See Also

reverseComplement, GENETIC_CODE, DNAString-class, RNAString-class, AAString-class, XStringSet-class, XStringViews-class, MaskedXString-class

Examples

```r
file <- system.file("extdata", "someORF.fa", package="Biostrings")
x <- read.DNAStringSet(file, "fasta")
x

## The first and last 1000 nucleotides are not part of the ORFs:
x <- DNAStringSet(x, start=1001, end=-1001)

## Before calling translate() on an ORF, we need to mask the introns
## if any. We can get this information from the SGD database
## (http://www.yeastgenome.org/).
## According to SGD, the 1st ORF (YAL001C) has an intron at 71..160
## (see http://db.yeastgenome.org/cgi-bin/locus.pl?locus=YAL001C)
```
y1 <- x[[1]]
mask1 <- Mask(length(y1), start=71, end=160)
masks(y1) <- mask1
y1
translate(y1)

## Codons
codons(y1)
which(width(codons(y1)) != 3)
codons(y1)[20:28]

---

**yeastSEQCHR1**  
*An annotation data file for CHR1 in the yeastSEQ package*

**Description**

This is a single character string containing DNA sequence of yeast chromosome number 1. The data were obtained from the Saccharomyces Genome Database (ftp://genome-ftp.stanford.edu/pub/yeast/data_download/sequence/genomic_sequence/chromosomes/fasta/).

**Details**

Annotation based on data provided by Yeast Genome project.

Source data built: Yeast Genome data are built at various time intervals. Sources used were downloaded Fri Nov 21 14:00:47 2003 Package built: Fri Nov 21 14:00:47 2003

**References**

http://www.yeastgenome.org/DownloadContents.shtml

**Examples**

data(yeastSEQCHR1)
nchar(yeastSEQCHR1)
Index

!=, BString, character-method (XString-class), 19
!=, XString, XString-method (XString-class), 19
!=, XString, XStringViews-method (XStringViews-class), 26
!=, XStringViews, XString-method (XStringViews-class), 26
!=, XStringViews, XStringViews-method (XStringViews-class), 26
!=, character, BString-method (XString-class), 19
!=, character, XStringViews-method (XStringViews-class), 26

*Topic category
alphabetFrequency, 32

*Topic character
stringDist, 72

*Topic classes
AAString-class, 1
AlignedXStringSet-class, 2
BOC_SubjectString-class, 4
DNAString-class, 4
InDel-class, 7
MaskedXString-class, 9
MIndex-class, 8
PairwiseAlignedFixedSubject-class, 14
PDict-class, 11
QualityScaledXStringSet-class, 16
RNAString-class, 18
XString-class, 19
XStringPartialMatches-class, 21
XStringQuality-class, 22
XStringSet-class, 23
XStringViews-class, 26

*Topic cluster
stringDist, 72

*Topic datasets

phiX174Phage, 64
substitution.matrices, 74
yeastSEQCHR1, 79

*Topic data
AMINO_ACID_CODE, 2
GENETIC_CODE, 5
IUPAC_CODE_MAP, 6
substitution.matrices, 74

*Topic manip
chartr, 35
gregexpr2, 38
injectHardMask, 39
maskMotif, 41
matchPWM, 55
readFASTA, 67
replaceLetterAt, 68
reverseComplement, 70
subXString, 73
translate, 77
XStringSet-io, 25

*Topic methods
AAString-class, 1
align-utils, 30
AlignedXStringSet-class, 2
BOC_SubjectString-class, 4
chartr, 35
DNAString-class, 4
findPalindromes, 36
InDel-class, 7
letter, 40
MaskedXString-class, 9
maskMotif, 41
match-utils, 43
matchLRPatterns, 47
matchPattern, 56
matchPDict, 48
matchPDict-inexact, 52
matchProbePair, 59
matchPWM, 55
MIndex-class, 8
needwunsQS, 61
PairwiseAlignedFixedSubject-class, 14
INDEX

pairwiseAlignment, 62
PDict-class, 11
pid, 65
pmatchPattern, 66
QualityScaledXStringSet-class, 16
reverseComplement, 70
RNAString-class, 18
subXString, 73
toComplex, 76
translate, 77
XString-class, 19
XStringPartialMatches-class, 21
XStringQuality-class, 22
XStringSet-class, 23
XStringViews-class, 26
XStringViews-constructors, 29

∗ Topic models
needwunsQS, 61

∗ Topic multivariate
stringDist, 72

∗ Topic utilities
AMINO_ACID_CODE, 2
GENETIC_CODE, 5
injectHardMask, 39
IUPAC_CODE_MAP, 6
matchPWM, 55
readFASTA, 67
replaceLetterAt, 68
substitution.matrices, 74
XStringSet-io, 25

.inplaceReplaceLetterAt
(replaceLetterAt), 68
==,BString,character-method
(XString-class), 19
==,XString,XString-method
(XString-class), 19
==,XString,XStringViews-method
(XStringViews-class), 26
==,XStringViews,XString-method
(XStringViews-class), 26
==,XStringViews,XStringViews-method
(XStringViews-class), 26
==,XStringViews,XStringViews-method
(XStringViews-class), 26
==,XStringViews,XStringViews-method
(XStringViews-class), 26
==,character,BString-method
(XString-class), 19
==,character,XStringViews-method
(XStringViews-class), 26

(PDict-class), 11
[,AlignedXStringSet,ANY,ANY,ANY-method
(AlignedXStringSet-class), 2
[,PairwiseAlignedFixedSubject,ANY,ANY,ANY-method
(PairwiseAlignedFixedSubject-class), 14
[,QualityAlignedXStringSet,ANY,ANY,ANY-method
(AlignedXStringSet-class), 2
[,QualityScaledXStringSet,ANY,ANY,ANY-method
(QualityScaledXStringSet-class), 16
[,XString,ANY,ANY,ANY-method
(XString-class), 19
[,XStringPartialMatches,ANY,ANY,ANY-method
(XStringPartialMatches-class), 21
[,XStringSet,ANY,ANY,ANY-method
(XStringSet-class), 23
[<-,AlignedXStringSet,ANY,ANY,ANY-method
(AlinedXStringSet-class), 2
[<-,PairwiseAlignedFixedSubject,ANY,ANY,ANY-method
(PairwiseAlignedFixedSubject-class), 14
[[,ByPos_MIndex-method
(MIndex-class), 8
[[,PDict-method(PDict-class), 11
[[,XStringSet-method
(XStringSet-class), 23
[[<-,XStringSet-method
(XStringSet-class), 23

AA_ALPHABET, 6
AA_ALPHABET(AAString-class), 1
AAString, 1, 2, 5, 6, 18, 19, 23, 78
AAString(AAString-class), 1
AAString-class, 1, 20, 24, 26, 75, 78
AAStringSet, 16
AAStringSet(XStringSet-class), 23
AAStringSet-class, 17
AAStringSet-class
(XStringSet-class), 23
ACTree(PDict-class), 11
ACTree-class(PDict-class), 11
ACTree2(PDict-class), 11
ACTree2-class(PDict-class), 11
adjacentViews
(XStringViews-constructors), 29
agrep, 73
align-utils, 30, 46
aligned
(AlignedXStringSet-class), 2
aligned, AlignedXStringSet-method
(AlignedXStringSet-class), 2
aligned, PairwiseAlignedFixedSubject-method
(PairwiseAlignedFixedSubject-class), 14
AlignedXStringSet
(AlignedXStringSet-class), 2
AlignedXStringSet-class, 16
AlignedXStringSet-class, 2, 31
alignScore (needwunsQS), 61
alphabet, 9
alphabet, AAString-method
(AAString-class), 1
alphabet, AlignedXStringSet-method
(AlignedXStringSet-class), 2
alphabet, BString-method
(BString-class), 19
alphabet, DNAString-method
(DNAString-class), 4
alphabet, MaskedXString-method
(MaskedXString-class), 9
alphabet, PairwiseAlignedFixedSubject-method
(PairwiseAlignedFixedSubject-class), 14
alphabet, RNAString-method
(RNAString-class), 18
alphabet, XString-method
(XString-class), 19
alphabet, XStringSet-method
(XStringSet-class), 23
alphabet, XStringViews-method
(XStringViews-class), 26
alphabetFrequency, 2, 5, 10, 18, 32, 35, 49, 58
alphabetFrequency, DNAString-method
(alphabetFrequency), 32
alphabetFrequency, DNAStringSet-method
(alphabetFrequency), 32
alphabetFrequency, MaskedXString-method
(alphabetFrequency), 32
alphabetFrequency, RNAString-method
(alphabetFrequency), 32
alphabetFrequency, RNAStringSet-method
(alphabetFrequency), 32
alphabetFrequency, XString-method
(alphabetFrequency), 32
alphabetFrequency, XStringSet-method
(alphabetFrequency), 32
alphabetFrequency, XStringViews-method
(alphabetFrequency), 32
alphabetFrequency, XStringViews-method
(alphabetFrequency), 32
alphabetFrequency, XStringViews-method
(alphabetFrequency), 32
as.character, AlignedXStringSet-method
(AlignedXStringSet-class), 2
as.character, AlignedXStringSet-method
(AlignedXStringSet-class), 2
as.character, DNAString-method
(DNAString-class), 4
as.character, MaskedXString-method
(MaskedXString-class), 9
as.character, PairwiseAlignedFixedSubject-method
(PairwiseAlignedFixedSubject-class), 14
as.character, XString-method
(XString-class), 19
as.character, XStringSet-method
(XStringSet-class), 23
as.character, XStringViews-method
(XStringViews-class), 26
as.complex, DNAString-method
(toComplex), 76
as.complex, DNAString-method
(toComplex), 76
as.integer, PhredQuality-method
(XStringQuality-class), 22
as.integer, SolexaQuality-method
(XStringQuality-class), 22
as.list, MTB_PDict-method
(PDict-class), 11
as.matrix, ACtree-method
(PDict-class), 11
as.matrix, PairwiseAlignedFixedSubject-method
(PairwiseAlignedFixedSubject-class), 14
as.matrix, XString-method
(XString-class), 19
as.matrix, XStringSet-method
(XStringSet-class), 23
as.matrix, XStringViews-method
(XStringViews-class), 26
as.numeric, PhredQuality-method
(XStringQuality-class), 22
as.numeric, SolexaQuality-method
(XStringQuality-class), 22
AMINO_ACID_CODE
(substitution.matrices), 74
BLOSUM100
(substitution.matrices), 74
BLOSUM45
(substitution.matrices), 74
BLOSUM50
(substitution.matrices), 74
BLOSUM62
(substitution.matrices), 74
INDEX

BLOSUM80 (substitution.matrices), 74
BOC2_SubjectString (BOC_SubjectString-class), 4
BOC2_SubjectString-class (BOC_SubjectString-class), 4
BOC_SubjectString (BOC_SubjectString-class), 4
BOC_SubjectString-class (BOC_SubjectString-class), 4
BSgenome, 69
BString, 1, 4, 5, 18, 22, 23, 31, 33, 34
BString (XString-class), 19
BString-class, 24, 26
BString-class (XString-class), 19
BStringSet, 16, 22, 31, 33
BStringSet (XStringSet-class), 23
BStringSet-class (XStringSet-class), 23
BStringViews (XStringSet-constructors), 29
BStringViews, ANY-method (XStringSet-constructors), 29
BStringViews, file-method (XStringSet-constructors), 29
BStringViews, XString-method (XStringSet-constructors), 29
BStringViews, XStringViews-method (XStringSet-constructors), 29
ByPos_MIndex-class (MIndex-class), 8

chartr, 35, 55, 69, 70
chartr, MaskedXString-method (chartr), 35
chartr, XString-method (chartr), 35
chartr, XStringSet-method (chartr), 35
chartr, XStringViews-method (chartr), 35
class:AAString (AAString-class), 1
class:AAStringSet (XStringSet-class), 23
class:ACtree (PDict-class), 11
class:ACtree2 (PDict-class), 11
class:AlignedXStringSet (AlignedXStringSet-class), 2
class:BOC2_SubjectString (BOC_SubjectString-class), 4
class:BOC_SubjectString-class (BOC_SubjectString-class), 4
class:BOC_SubjectString (BOC_SubjectString-class), 4
class:BSgenome, 69
class:BSString (XString-class), 19
class:BSStringSet (XStringSet-class), 23
class:ByPos_MIndex (MIndex-class), 8
class:DNAString (DNAString-class), 4
class:DNAStringSet (XStringSet-class), 23
class:InDel (InDel-class), 7
class:MaskedAAString (MaskedXString-class), 9
class:MaskedBString (MaskedXString-class), 9
class:MaskedDNAString (MaskedXString-class), 9
class:MaskedRNAString (MaskedXString-class), 9
class:MaskedXString (MaskedXString-class), 9
class:MIndex (MIndex-class), 8
class:MTB_PDict (PDict-class), 11
class:PairwiseAlignedFixedSubject (PairwiseAlignedFixedSubject-class), 14
class:PairwiseAlignedFixedSubjectSummary (PairwiseAlignedFixedSubject-class), 14
class:PDict (PDict-class), 11
class:PDict3Parts (PDict-class), 11
class:PhredQuality (XStringQuality-class), 11
class:PreprocessedTB (PDict-class), 11
class:QualityAlignedXStringSet (AlignedXStringSet-class), 2
class:QualityScaledAAStringSet
(QualityScaledXStringSet-class), (XStringQuality-class), 22
16
c::QualityScaledBStringSet
c::QualityScaledDNAStringSet
16
c::QualityScaledRNAStringSet
c::QualityScaledXStringSet-class, 16
c::RNAString
18
c::RNAStringSet
23
c::SolexaQuality
(XStringQuality-class), 22
c::TB_PDict
(PDict-class), 11
c::Twobit
(PDict-class), 11
c::XString(XString-class), 19
c::XStringPartialMatches
(XStringPartialMatches-class), 21
c::XStringQuality
(XStringQuality-class), 22
c::XStringSet
(XStringSet-class), 23
c::XStringViews
(XStringViews-class), 26
codons (translate), 77
codons, DNAString-method
(translate), 77
codons, MaskedDNAString-method
(translate), 77
codons, MaskedRNAString-method
(translate), 77
codons, RNAString-method
(translate), 77
codons, AAString-method
(translate), 77
c::AAString
(MaskedAAString-class), 9
c::BString
(MaskedBString-class), 9
c::PhredQuality
(XStringQuality-class), 22
c::SolexaQuality
(XStringQuality-class), 22
c::BStringSet
(XStringQuality-class), 22
c::BStringSet
(XStringQuality-class), 22
c::BStringSet
(SolexaQuality-method)
coerce, character, PhredQuality-method
coerce, character, SolexaQuality-method
coerce, integer, PhredQuality-method
coerce, integer, SolexaQuality-method
coerce, MaskedAAString
(AAString-method)
coerce, MaskedBString
(BString-method)
coerce, MaskedDNAString
(DNAString-method)
coerce, MaskedRNAString
(RNAString-method)
coerce, MaskedXString
(MaskCollection-method)
coerce, MaskedXString
(NormalIRanges-method)
coerce, MaskedXString
(XStringViews-method)
coerce, PhredQuality
(integer-method)
coerce, PhredQuality
(numeric-method)
coerce, RNAString
(MaskedRNAString-method)
coerce, SolexaQuality
(integer-method)
coerce, SolexaQuality
(numeric-method)
coerce, XStringSet
(AAStringSet-method)
coerce, XStringSet
(BStringSet-method)
coerce, XStringSet
(DNAStringSet-method)
coerce, XStringSet
(RNAStringSet-method)
coerce, XStringSet
(MaskCollection-method)
coerce, XStringSet
(NormalIRanges-method)
coerce, XStringSet
(XStringViews-method)
coerce, align-utils
(compareStrings)
coerce, character
(compareStrings)
coerce
(compareStrings)
INDEX

deletion, InDel-method
(InDel-class), 7

dinucleotideFrequency
(alphabetFrequency), 32
dist, 73
dna2rna (translate), 77
DNA_ALPHABET, 14, 23, 24
DNA_ALPHABET (DNAString-class), 4
DNAString, 1, 6, 7, 11–13, 18–20, 23, 27, 32, 33, 37, 44, 47, 49, 55, 60, 69, 70, 76–78
DNAString (DNAString-class), 4
DNAString-class, 4, 18, 20, 22, 24, 26, 38, 49, 56, 70, 75, 78
DNAStringSet, 11–13, 16, 32, 70, 78
DNAStringSet (XStringSet-class), 23
DNAStringSet-class, 14, 17, 70
DNAStringSet-class (XStringSet-class), 23
duplicated, PDict-method
(PDict-class), 11
duplicated, PreprocessedTB-method
(PDict-class), 11
end, AlignedXStringSet-method
(Ali neoXStringSet-class), 2
endIndex (MIndex-class), 8
endIndex, ByPos_MIndex-method
(MIndex-class), 8	errorSubstitutionMatrices
(substitution.matrices), 74
extractAll Matches (matchPDict), 48
fasta.info, 26
fasta.info (readFASTA), 67
FASTArecordsToBStringViews
(XStringSet-io), 25
FASTArecordsToCharacter
(XStringSet-io), 25
FASTArecordsToXStringViews
(XStringSet-io), 25
findComplementedPalindromes
(findPalindromes), 36
findComplementedPalindromes, DNAString-method
(findPalindromes), 36
findComplementedPalindromes, DNAString-set
(findPalindromes), 36
findComplementedPalindromes, MaskedXString-method
(MaskedXString-class), 4
findComplementedPalindromes, MaskedXString-set
(MaskedXString-class), 4
findComplementedPalindromes, XString-method
(findPalindromes), 36
findComplementedPalindromes, XString-set
(findPalindromes), 36
findPalindromes, 36, 48, 60, 70
findPalindromes, MaskedXString-method
(findPalindromes), 36
findPalindromes, XString-method
(findPalindromes), 36
findPalindromes, XStringViews-method
(findPalindromes), 36
gaps, 28
gaps, MaskedXString-method
(MaskedXString-class), 9
GENETIC_CODE, 2, 5, 34, 78
gregexpr, 39
gregexpr2, 38
hasOnlyBaseLetters
(alphabetFrequency), 32
hasOnlyBaseLetters, DNAString-method
(alphabetFrequency), 32
hasOnlyBaseLetters, DNAString-set
(alphabetFrequency), 32
hasOnlyBaseLetters, MaskedDNAString-method
(alphabetFrequency), 32
hasOnlyBaseLetters, MaskedRNAString-method
(alphabetFrequency), 32
hasOnlyBaseLetters, RNAString-method
(alphabetFrequency), 32
hasOnlyBaseLetters, RNAString-set
(alphabetFrequency), 32
head, PDict3Parts-method
(PDict-class), 11
head, TB_PDict-method
(PDict-class), 11
InDel (InDel-class), 7
indel (AlignedXStringSet-class), 2
indel, AlignedXStringSet-method
(AlignedXStringSet-class), 2
InDel-class, 7
initialize, ACtree-method
(PDict-class), 11
initialize, ACtree2-method
(PDict-class), 11
initialize, AlignedXStringSet-method
(AlignedXStringSet-class),
initialize, BOC2_SubjectString-method
(BOC_SubjectString-class),
initialize, BOC_SubjectString-set
(BOC_SubjectString-class),
initialize, PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class), 14
initialize, PreprocessedTB-method (PDict-class), 11
initialize, QualityAlignedXStringSet-method (AlignedXStringSet-class), 2
initialize, Twobit-method (PDict-class), 11
injectHardMask, 10, 39, 69
injectHardMask, MaskedXString-method (injectHardMask), 39
injectHardMask, XStringViews-method (injectHardMask), 39
injectSNPs, 69
insertion (InDel-class), 7
insertion, InDel-method (InDel-class), 7
IRanges, 8, 23, 24, 45, 70
IRanges-class, 9, 46, 70
IRanges-utils, 10
isMatching, 49
isMatching (match-utils), 43
isMatchingAt (match-utils), 57, 58
isMatchingAt (match-utils), 43
isMatchingEndingAt (match-utils), 43
isMatchingEndingAt (match-utils), 43
isMatchingEndingAt, character-method (match-utils), 43
isMatchingEndingAt, XString-method (match-utils), 43
isMatchingStartingAt (match-utils), 43
isMatchingStartingAt, character-method (match-utils), 43
isMatchingStartingAt, XString-method (match-utils), 43
isMatchingStartingAt, XString-method (match-utils), 43
IUPAC_CODE_MAP, 4, 5, 6, 18, 44, 46, 47, 69
lcprefix (pmatchPattern), 66
lcsubstr (XString, character-method (pmatchPattern), 66
lcsuffix, character, character-method (pmatchPattern), 66
lcsuffix, character, XString-method (pmatchPattern), 66
lcsuffix, XString, character-method (pmatchPattern), 66
lcsuffix, XString, XString-method (pmatchPattern), 66
lcsubstr, XString, character-method (pmatchPattern), 66
lcsubstr, character, character-method (pmatchPattern), 66
mask (maskMotif), 41
MaskCollection, 9, 45, 70
MaskCollection-class, 10, 42, 46, 70
MaskedAAString, 39
INDEX

MIndex (MIndex-class), 8
MIndex-class, 8, 28, 46, 49, 53, 58
mismatch, 58
mismatch (match-utils), 43
mismatch, AlignedXStringSet, missing-method (align-utils), 30
mismatch, ANY, XString-method (match-utils), 43
mismatchSummary (align-utils), 30
mismatchSummary, AlignedXStringSet-method (align-utils), 30
mismatchSummary, PairwiseAlignedFixedSubject-method (match-utils), 43
mismatchSummary, QualityAlignedXStringSet-method (align-utils), 30
mismatchTable (align-utils), 30
mismatchTable, AlignedXStringSet-method (match-utils), 43
mismatchTable, PairwiseAlignedFixedSubject-method (match-utils), 43
mismatchTable, QualityAlignedXStringSet-method (match-utils), 43
mkAllStrings (alphabetFrequency), 32
MTB_PDict (PDict-class), 11
MTB_PDict-class (PDict-class), 11
names, MIndex-method (MIndex-class), 8
names, PDict-method (PDict-class), 11
names, XStringSet-method (XStringSet-class), 23
names<-, MIndex-method (MIndex-class), 8
names<-, PDict-method (PDict-class), 11
names<-, XStringSet-method (XStringSet-class), 23
narrow, 23, 24
narrow, QualityScaledXStringSet-method (QualityScaledXStringSet-class), 16
narrow, XStringSet-method (XStringSet-class), 23
nchar, AlignedXStringSet-method (AlignedXStringSet-class), 2
nchar, MaskedXString-method (MaskedXString-class), 9
nchar, PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class), 14
nchar, PairwiseAlignedFixedSubjectSummary-method (PairwiseAlignedFixedSubjectSummary-class), 14
nmatch (match-utils), 43
nmatch, ANY, XStringViews-method (match-utils), 43
nmatch, PairwiseAlignedFixedSubject, missing-method (align-utils), 30
nmatch, PairwiseAlignedFixedSubjectSummary, missing-method (align-utils), 30
nnmismatch (match-utils), 43
nnmismatch, AlignedXStringSet, missing-method

89
INDEX

(alignment-utils), 30
nmismatch, ANY, XStringViews-method
(match-utils), 43
nmismatch, PairwiseAlignedFixedSubject, missing-
(alignment-utils), 30
nmismatch, PairwiseAlignedFixedSubjectSummary-
(alignment-utils), 30
nmismatchEndingAt (match-utils), 43
nmismatchStartingAt (match-utils), 43
NormalIRanges, 70
NormalIRanges-class, 70
nucleotideSubstitutionMatrix
(substitution.matrices), 74
oligonucleotideFrequency
(alphabetFrequency), 32
oligonucleotideFrequency, DNAString-method
(alphabetFrequency), 32
oligonucleotideFrequency, MaskedXString-method
(alphabetFrequency), 32
oligonucleotideFrequency, RNAString-method
(alphabetFrequency), 32
oligonucleotideFrequency, XStringViews-method
(alphabetFrequency), 32
oligonucleotideTransitions
(alphabetFrequency), 32
order (XStringSet-class), 23
order, XStringSet-method
(XStringSet-class), 23
PairwiseAlignedFixedSubject, 63, 65
PairwiseAlignedFixedSubject
(PairwiseAlignedFixedSubject-class), (findPalindromes), 36
palindromeArmLength
(findPalindromes), 36
palindromeArmLength, XString-method
(findPalindromes), 36
palindromeArmLength, XStringViews-method
(findPalindromes), 36
palindromeLeftArm
(findPalindromes), 36
palindromeLeftArm, XString-method
(findPalindromes), 36
palindromeRightArm
(findPalindromes), 36
palindromeRightArm, XStringViews-method
(findPalindromes), 36
pairwiseAlignment, character, character-method
(pairwiseAlignment), 62
pairwiseAlignment, character, QualityScaledXString-
(pairwiseAlignment), 62
pairwiseAlignment, character, XString-method
(pairwiseAlignment), 62
pairwiseAlignment, character, XStringSet-method
(pairwiseAlignment), 62
pairwiseAlignment, QualityScaledXStringSet, character-
(pairwiseAlignment), 62
pairwiseAlignment, QualityScaledXStringSet, Quality-
(pairwiseAlignment), 62
pairwiseAlignment, QualityScaledXStringSet, XString-
(pairwiseAlignment), 62
pairwiseAlignment, QualityScaledXStringSet, XStringSet-
(pairwiseAlignment), 62
pairwiseAlignment, XString, character-method
(pairwiseAlignment), 62
pairwiseAlignment, XString, QualityScaledXString-
(pairwiseAlignment), 62
pairwiseAlignment, XString, XString-method
(pairwiseAlignment), 62
pairwiseAlignment, XString, XStringSet-method
(pairwiseAlignment), 62
pairwiseAlignment, XStringSet, character-method
(pairwiseAlignment), 62
pairwiseAlignment, XStringSet, QualityScaledXString-
(pairwiseAlignment), 62
pairwiseAlignment, XStringSet, XString-method
(pairwiseAlignment), 62
pairwiseAlignment, XStringSet, XStringSet-method
(pairwiseAlignment), 62
PAM120
(substitution.matrices), 74
PAM250
(substitution.matrices), 74
pairwiseAlignment, 4, 8
pairwiseAlignment, 4, 8, 16, 22, 31, 57, 58, 61, 62, 66, 73, 75
PAM120
(substitution.matrices), 74
PAM250
(substitution.matrices), 74
}
INDEX

PAM30 (substitution.matrices), 74
PAM40 (substitution.matrices), 74
PAM70 (substitution.matrices), 74

pattern
(XStringPartialMatches-class), 21

pattern, PairwiseAlignedFixedSubject-method
(PairwiseAlignedFixedSubject-class), 2

pattern, XStringPartialMatches-method
(XStringPartialMatches-class), 21

patternFrequency (PDict-class), 11
patternFrequency, PDict-method
(PDdict-class), 11

PDdict, 49, 53
PDdict (PDdict-class), 11
PDdict, AsIs-method (PDdict-class), 11

PDdict, character-method
(PDdict-class), 11
PDdict, DNAStringSet-method
(PDdict-class), 11
PDdict, XStringViews-method
(PDdict-class), 11
PDdict-class, 9, 11, 49, 53
PDdict3Parts (PDdict-class), 11
PDdict3Parts-class (PDdict-class), 11

phiX174Phage, 64

PhredQuality
(XStringQuality-class), 22
PhredQuality-class, 75
PhredQuality-class
(XStringQuality-class), 22

pid, 65

pid, PairwiseAlignedFixedSubject-method
(pid), 65

pmatchPattern, 66
pmatchPattern, character-method
(pmatchPattern), 66
pmatchPattern, XString-method
(pmatchPattern), 66
pmatchPattern, XStringViews-method
(pmatchPattern), 66

PreprocessedTB (PDdict-class), 11
PreprocessedTB-class
(PDdict-class), 11

print.needwunsQS (needwunsQS), 61

PWMscore (matchPWM), 55

quality
(QualityScaledXStringSet-class), read.DNAStringSet, 68
read.DNAStringSet (XStringSet-io), 25
read.Mask, 42
read.RNAStringSet, 68
read.RNAStringSet (XStringSet-io), 25
read.table, 68
read.XStringViews (XStringSet-io), 25
readFasta, 25, 26, 67
read,MaskedXString-method (MaskedXString-class), 9
rep,AlignedXStringSet-method (AlignedXStringSet-class), 2
rep,PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class),
  14
rep,XStringSet-method (XStringSet-class), 23
replaceLetterAt (replaceLetterAt), 68
replaceLetterAt,DNAString-method (replaceLetterAt), 68
replaceLetterAtLoc (replaceLetterAt), 68
rev, 34
reverse,MaskedXString-method, 10
reverse,MaskedXString-method (reverseComplement), 70
reverse,XString-method, 20, 34
reverse,XString-method (reverseComplement), 70
reverse,XStringSet-method (reverseComplement), 70
reverse,XStringViews-method (reverseComplement), 70
reverseComplement, 5, 18, 35, 48, 56, 60, 70, 78
reverseComplement,DNAString-method (reverseComplement), 70
reverseComplement,DNAStringSet-method (reverseComplement), 70
reverseComplement,MaskedDNAString-method (reverseComplement), 70
reverseComplement,MaskedRNAString-method (reverseComplement), 70
reverseComplement,matrix-method (matchPWM), 55
reverseComplement,RNAString-method (reverseComplement), 70
reverseComplement,RNAStringSet-method show, Twobit-method (PDict-class),
  11
reverseComplement,XStringViews-method (reverseComplement), 70
rnav2dna (translate), 77
RNA_ALPHABET (RNAString-class), 18
RNA_BASES (RNAString-class), 18
RNA_GENETIC_CODE (GENETIC_CODE), 5
RNAString, 1, 5–7, 19, 20, 23, 27, 32, 33, 44, 47, 70, 78
RNAString (RNAString-class), 18
RNAString-class, 5, 18, 20, 24, 26, 70, 78
RNAStringSet, 16, 32, 70, 78
RNAStringSet (XStringSet-class), 23
RNAStringSet-class, 17, 70
RNAStringSet-class (XStringSet-class), 23
scan, 68
score (PairwiseAlignedFixedSubject-class), 14
score,PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class),
  14
score,PairwiseAlignedFixedSubjectSummary-method (PairwiseAlignedFixedSubject-class),
  14
show,ACTree-method (PDict-class), 11
show,ACTree2-method (PDict-class), 11
show,AlignedXStringSet-method (AlignedXStringSet-class), 2
show,ByPos_MIndex-method (MIndex-class), 8
show,MaskedXString-method (MaskedXString-class), 9
show,MTB_PDict-method (PDict-class), 11
show,PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class),
  14
show,PairwiseAlignedFixedSubjectSummary-method (PairwiseAlignedFixedSubject-class),
  14
show,QualityScaledXStringSet-method (QualityScaledXStringSet-class),
  16
show,TB_PDict-method (PDict-class), 11
INDEX

show,XString-method (XString-class), 19
show,XStringPartialMatches-method (XStringPartialMatches-class), 21
show,XStringSet-method (XStringSet-class), 23
show,XStringViews-method (XStringViews-class), 26
SolexaQuality (XStringQuality-class), 22
SolexaQuality-class, 75
SolexaQuality-class (XStringQuality-class), 22
sort,XStringSet-method (XStringSet-class), 23
srPhiX174 (phiX174Phage), 64
start,AlignedXStringSet-method (AlignedXStringSet-class), 2
startIndex (MIndex-class), 8
startIndex,ByPos_MIndex-method (MIndex-class), 8
stringDist, 63, 72
stringDist,character-method (stringDist), 72
stringDist,QualityScaledXStringSet-method (stringDist), 72
stringDist,XStringSet-method (stringDist), 72
strrev, 70
strrev(alphabetFrequency), 32
strsplit, 34
subBString (subXString), 73
subject,PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class), 14
subpatterns (XStringPartialMatches-class), 21
subpatterns,XStringPartialMatches-method (XStringPartialMatches-class), 21
subseq, 20, 41, 74
subseq,MaskedXString-method (MaskedXString-class), 9
substitution.matrices, 61, 63, 73, 74
substr, 41
substr (subXString), 73
substr,MaskedXString-method (subXString), 73
substr,XString-method
(subXString), 73
substring, 41
substring (subXString), 73
substring,MaskedXString-method (subXString), 73
substring,XString-method (subXString), 73
substring,XString-method
(subXString), 73
summary,PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class), 14
tail,PDict3Parts-method (PDict-class), 11
tail,TB_PDict-method (PDict-class), 11
tb(PDict-class), 11
tb,PDict3Parts-method (PDict-class), 11
tb,PreprocessedTB-method (PDict-class), 11
tb,TB_PDict-method (PDict-class), 11
tb.width(PDict-class), 11
tb.width,PDict3Parts-method (PDict-class), 11
tb.width,PreprocessedTB-method (PDict-class), 11
tb.width,TB_PDict-method (PDict-class), 11
TB_PDict-class (PDict-class), 11
TB_PDict-class (PDict-class), 11
toComplex, 76
toComplex,MaskedXString-method (toComplex), 76
toComplex,DNAString-method (toComplex), 76
toString,AlignedXStringSet-method (AlignedXStringSet-class), 14
toString,MaskedXString-method (MaskedXString-class), 9
toString,PairwiseAlignedFixedSubject-method (PairwiseAlignedFixedSubject-class), 14
toString,XString-method (XString-class), 19
toString,XStringSet-method (XStringSet-class), 23
toString,XStringViews-method (XStringViews-class), 26
transcribe (translate), 77
translate, 77
translate,DNAString-method (translate), 77
INDEX

width, PDict-method (PDict-class), 11
width, PDict3Parts-method (PDict-class), 11
width, PreprocessedTB-method (PDict-class), 11
width, XStringSet-method (XStringSet-class), 23
write.BStringViews (XStringSet-io), 25
write.table, 68
write.XStringSet (XStringSet-io), 25
write.XStringViews (XStringSet-io), 25
writeFASTA, 26
writeFASTA (readFASTA), 67
wtPhiX174 (phiX174Phage), 64
XRleInteger, 45, 46
XString, 1, 4, 8, 9, 17, 18, 23, 24, 26, 27, 29, 30, 33–37, 40, 41, 44, 47, 49, 56, 61, 62, 67, 69, 70, 73, 74
XString (XString-class), 19
XString, AsIs-method (XString-class), 19
XString, character-method (XString-class), 19
XString, MaskedXString-method (MaskedXString-class), 9
XString, XString-method (XString-class), 19
XString-class, 16
XString-class, 2, 5, 10, 18, 19, 21, 28, 30, 31, 34, 35, 41, 42, 46, 48, 67, 74
XStringPartialMatches-class, 21
XStringQuality, 16, 17, 62, 72
XStringQuality (XStringQuality-class), 22
XStringQuality-class, 17, 22, 63
XStringSet, 17, 19, 23, 25, 26, 33, 35, 56, 62, 70, 72
XStringSet (XStringSet-class), 23
XStringSet, AsIs-method (XStringSet-class), 23
XStringSet, character-method (XStringSet-class), 23
XStringSet, XString-method (XStringSet-class), 23
XStringSet, XStringSet-method (XStringSet-class), 23
XStringSet, XStringViews-method (XStringSet-class), 23
XStringSet-class, 4
XStringSet-class, 20, 23, 26, 28, 31, 34, 35, 78
XStringSet-jo, 25
XStringSetToFASTAreviews (XStringSet-jo), 25
XStringViews, 8, 10–12, 17, 23, 25, 26, 29, 31–33, 35–37, 39–42, 45–47, 55–57, 60, 70, 73, 78
XStringViews (XStringViews-constructors), 29
XStringViews, ANY-method (XStringViews-constructors), 29
XStringViews, XString-method (XStringViews-constructors), 29
XStringViews, XStringViews-method (XStringViews-constructors), 29
XStringViews-class, 16
XStringViews-class, 9, 10, 14, 20, 21, 24, 26, 28, 30, 31, 34, 35, 38, 40–42, 46, 48, 49, 56, 58, 67, 70, 74, 78
XStringViews-constructors, 28, 29
yeastSEQCHR1, 79